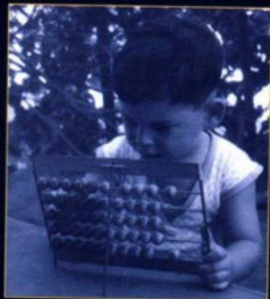
Broadview®
WWW.BROADVIEW.COM.CN

The Elements of Computing Systems

Building a Modern Computer from First Principles



Noam Nisan and Shimon Schocken

The Elements of Computing Systems

Building a Modern Computer from First Principles

计算机系统要素

从 零 开 始 构 建 现 代 计 算 机

[美] Noam Nisan Shimon Schocken 著
周维 宋磊 陈曦 译 刘天田 审校

電子工業出版社
PHEI
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE Caliente.com

计算机系统要素

——从零开始构建现代计算机

The Elements of Computing Systems

Building a Modern Computer from First Principles

[美] Noam Nisan
Shimon Schocken 著

周 维 宋 磊 陈 曦 译

刘天田 审校

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内容简介

本书通过展现简单但功能强大的计算机系统之构建过程，为读者呈现了一幅完整、严格的计算机应用科学大图景。本书作者认为，理解计算机工作原理的最好方法就是亲自动手，从零开始构建计算机系统。

通过12个章节和项目来引领读者从头开始，本书逐步地构建一个基本的硬件平台和现代软件阶层体系。在这个过程中，读者能够获得关于硬件体系结构、操作系统、编程语言、编译器、数据结构、算法以及软件工程的详实知识。通过这种逐步构造的方法，本书揭示了计算机科学知识中的重要成分，并展示其它课程中所介绍的理论和应用技术如何融入这幅全局大图景当中去。全书基于“先抽象再实现”的阐述模式，每一章都介绍一个关键的硬件或软件抽象，一种实现方式以及一个实际的项目。完成这些项目所必要的计算机科学知识在本书中都有涵盖，只要求读者具备程序设计经验。本书配套的支持网站提供了书中描述的用于构建所有硬件和软件系统所必需的工具和资料，以及用于12个项目的200个测试程序。

全书内容广泛、涉猎全面，适合计算机及相关专业本科生、研究生、技术开发人员、教师以及技术爱好者参考和学习。

© 2005 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

CHINESE SIMPLIFIED language edition published by PUBLISHING HOUSE OF ELECTRONICS INDUSTRY, Copyright © 2006

本书中文简体版专有出版权由MIT Press授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号：图字：01-2006-3844

图书在版编目（CIP）数据

计算机系统要素：从零开始构建现代计算机 / （美）尼萨（Nisan,N.），（美）锡肯（Schocken,S.）著；周维，宋磊，陈曦译. —北京：电子工业出版社，2007.2

书名原文：The Elements of Computing Systems: Building a Modern Computer from First Principles

ISBN 978-7-121-03336-0

Ⅰ.计… Ⅱ.①尼…②锡…③周…④宋…⑤陈… Ⅲ.计算机系统 Ⅳ.TP30

中国版本图书馆CIP数据核字（2006）第125896号

责任编辑：周 筠

印　　刷：北京智力达印刷有限公司

装　　订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱　邮编　100036

开　　本：787×980　1/16　印张：22.25　字数：380千字

印　　次：2007年2月第1次印刷

定　　价：45.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：（010）68279077；邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

关键术语中英对照表

本书涉及的专业领域较为广泛，在阅读过程中会遇到各个领域的许多专业术语。下表列出了本书用到的主要术语的英文原文以及对应的中文译文，供读者参考和辨析（本书正文中出现的专业术语在必要时都标注了原文，以便于读者参考和使用索引）。

英文术语	中文译法
allocation	（内存）分配
assembly	汇编；汇编语言
assembler	汇编编译器
class	类
class type	“类”类型
compiler	编译器
constant	常量
constructor	构造函数
de-allocation	（内存）去配
dispose	清除
field	域；字段
function	函数；功能
identifier	标识符
invoke	唤起（执行）
label	标签
method	方法
object	对象
object-oriented	面向对象（的）
object-based	基于对象的
object instance	对象实例

英文术语	中文译法
parse/ parsing	（语法）分析
parser	（语法）分析器
parse table	（语法）分析表
parse tree	语法分析树；分析树
pointer	指针
procedure	过程
procedural	过程化（的）
reference	引用
subroutine	子程序
symbol	符号
syntax	语法
syntax tree	语法树
syntax analysis	语法分析
syntax analyzer	语法分析器
token	字元
tokenize	字元化
tokenizer	字元转换器
type conversion	类型转换
variable	变量

技术编辑尽了最大努力来保持术语中文译文的明确性和限定性，但仍可能难免存在疏漏之处，欢迎广大读者批评指正。

若您对本书在翻译方面的任何技术议题存在疑问，或对本书的翻译质量有任何意见、建议和批评，都欢迎您写信给本书的技术编辑（*fangzhou@broadview.com.cn*）。博文视点欢迎并感谢读者指出本书中存在的任何错误，更欢迎大家对博文视点出版的任何图书提出意见、建议和批评。感谢您对博文视点的支持！祝您读书愉快。

本书技术编辑　方　舟

电子邮件地址：*fangzhou@broadview.com.cn*

2007 年 1 月

前言
Preface

What I hear, I forget; What I see, I remember; What I do, I understand.

耳听为虚；眼见为实；实践出真知。

不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之；学至于行之而止矣。[1]

——孔子（公元前 551 ~ 公元前 479 BC）

过去，凡是计算机专业人员都对计算机的工作原理和工作方式了如指掌。计算机体系中的硬件、软件、编译器以及操作系统之间的交互既简单又透明，因此要把握计算机系统大局观并非难事。然而随着现代计算机技术的日趋复杂，这种明晰性不复存在：计算机科学领域里面大多数基本思想和技术都被隐藏在众多抽象接口以及私有实现的层面之下。这种复杂性导致了无法避免的结果，即领域专业化；这使得多门计算机科学领域应运而生，每个领域只涵盖整个学科中的某一个方面。

之所以要编写这本书，正是因为本书作者有感于：很多学习计算机科学的学生识木而不知林，疲于埋头学习程序设计、各种理论以及工程知识，却失去了对计算机系统整体的把握和理解，未曾停下来欣赏计算机系统大局观的美景。这个大局观为我们展示的是：硬件系统和软件系统如何经由隐藏的抽象、接口以及基于各种约定的实现所编织起来的网，从而紧紧地关联在一起。由于没有从表及里地透彻领略这个繁复大局观的魅力，使很多学生和计算机从业人员产生了不安的感觉，因为他们并没有完全透彻理解和掌握计算机的内部工作原理。

本书作者相信，理解计算机工作原理的最好方法就是亲自动手，从零开始构建计算机系统。由此我们提出了如下的概念：描述一种简单但功能较强的计算机系统，让学生从最基本的逻辑门开始构建其硬件平台和软件层级。要做就要把事情做好。之所以这么说是因

[1] 这句话出自荀子《儒效篇》。——审校者

为，从头开始构建完整的通用计算机系统是个艰巨的任务。因此，本书不仅阐述如何构建计算机系统，而且让读者亲自参与实践，了解如何有效地计划和管理大规模硬件/软件开发项目。此外，我们从最原始最基本的构建模块开始，通过递归向上和逻辑推理等手段，展示如何构建复杂且有用的系统。

范围
Scope

本书通过一系列的硬件和软件实践项目，向读者展示计算机科学知识中的大部分核心内容。这些实践项目将会为您展示计算机科学中的理论知识和应用技术是如何应用于工程实践中的。本书涵盖如下主题：

- **硬件**：逻辑门；布尔运算； multiplexor（多路复用器）；触发器（flip-flop）；寄存器（register）；RAM 单元；计数器；硬件描述语言（HDL，Hardware Description Language）；芯片的仿真及测试。
- **体系架构**：ALU/CPU 的设计与实现；机器代码；汇编语言程序设计；取址模式；I/O 内存映像。
- **操作系统**：内存管理；数学计算程序库；基本 I/O 驱动程序；屏幕管理；文件 I/O；对高级语言的支持。
- **程序设计语言**：基于对象（object-based）的设计和编程模式；抽象数据类型；作用域；语法和语义；引用（reference）机制。
- **编译器**：词法分析；自顶向下的语法分析；符号表（symbol table）；基于堆栈（stack-based）的虚拟机；代码生成；数组和对象的实现。
- **数据结构和算法**：堆栈；哈希表；链表；递归；算术算法；几何算法；运行效率。
- **软件工程**：模块化设计；接口/实现范式；API 设计和文档；主动式测试（即极限编程理论中的单元测试等）；广义的程序设计概念；质量保证体系。

介绍这些主题的目的只有一个：从零开始构建现代计算机系统。这同时也是我们选择讨论主题时所遵循的原则，即本书集中讨论构建具有完整功能的计算机系统所需要的最小集合。因此，该最小集合中包含了很多应用计算机科学中的基本思想。

课程
Course

本书的读者对象主要是高等院校的计算机科学以及其他工程学科的本科生和研究生。以本书为基础的课程与普通计算机科学课程是“纵向交叉”的，学生在普通课程中的任何时候都可以学习本书的内容。本书有两个明显的课程切入点，一个是“CS-2”（即学完程序设计之后），一个是“CS-199”（即学完所有相关课程之后，作为总结性的综合课程）。“CS-2”课程可作为面向系统的计算机科学入门，而“CS-199”课程可作为全面的、面向项目的系统构建课程。此类课程可称为计算机科学的系统结构，或计算系统要素、数字系统构建、计算机系统构建，或是“让我们来构建计算机”等等。根据涵盖的主题以及教学进度计划，本书可以在一个学期内讲授，也可以分为两个学期进行讲授。

本书是完全与其他课程独立的，所需的预备知识仅仅是基本的编程能力（任何语言）。因此，本书不仅适用于计算机科学专业的学生，而且也适用于计算机实用技术专业的学生（以此学习硬件结构、操作系统以及现代软件工程的基本知识）。对于任何技术或科学专业的学生，只要学完一门程序设计课程之后，就可以利用本书以及本书配套的支持网站作为自学教程。

本书的结构
Structure

在开始的简介内容里，我们提出了构建的方法并预览了本书所讨论的主要硬件/软件抽象。这为第 1 至 12 章的阐述奠定了基础。从第 1 章到第 12 章，每章都要讨论三个内容：1）一种关键的硬件抽象或软件抽象；2）该抽象的实现原理；3）一个实际项目，用来构建并测试所构建的系统。前 5 章主要讨论简单的现代计算机硬件平台的构建。第 6 至 12 章描述一个典型的多层（multi-tier）软件阶层体系的设计与实现，包括构建一门基于对象（object-based）的程序设计语言和一个简单的操作系统。图 P.1 展示了完整的构建计划。

本书遵循“**先抽象再实现**”的方式，每一章开头先有**背景知识**部分，用来描述相关概念和通用的硬件或者软件系统。接下来的**规范详述**部分提供了对系统抽象的明确描述，即

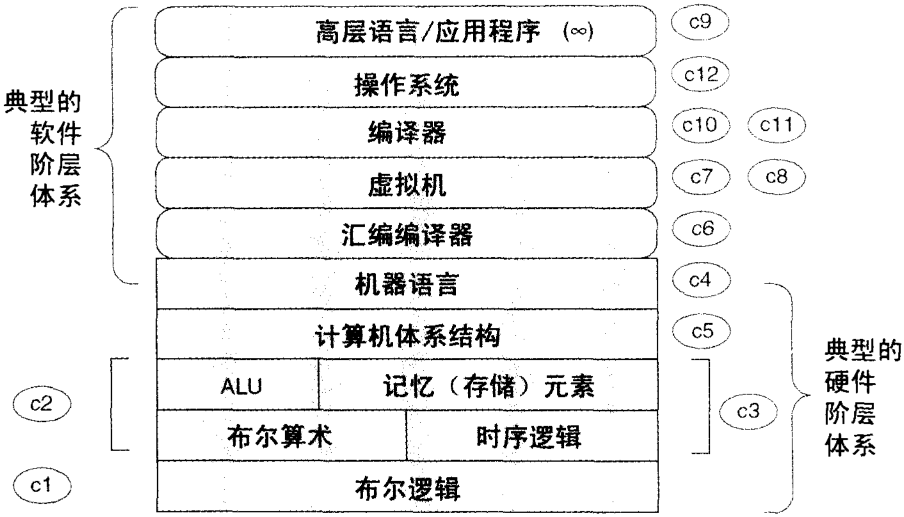


图 P.1　本书及课程概览，圆圈中显示了涉及到的章节

它能提供何种服务。再接下来的**实现**部分继续讨论如何来实现抽象。之后的**观点**部分强调每章当中没有考虑却又值得注意的一些问题。每章的最后都是**项目**部分，提供了逐步进行构建的说明、测试程序和用于实际进行构建并进行单元测试的软件工具。

项目
Projects

本书介绍的计算机系统是实际可用的系统，可被实际地构建出来，并能够正常运作。只要读者跟随本书的脚步，逐步构建出这个计算机系统，就会对计算机系统有更透彻的理解，这与只阅读而不动手实践所达到的学习效果相比有天壤之别。因此，本书在内容的编排和选择上着重针对那些卷起袖子准备动手实践，从零开始构建计算机系统的读者。

每一章都完整阐述了一个单独的硬件或软件开发项目。用来构建计算机系统硬件平台的前四个项目（第 1 至 4 章的项目），是利用一个简单的硬件描述语言（HDL）来实现的，并可在与本书配套的硬件仿真器上进行模拟测试。后续第 5 至 9 章的 5 个项目（即汇编编

译器，虚拟机第I部分和虚拟机第II部分，以及编译器第I部分和编译器第II部分）可以采用任何现代编程语言来编写。最后第10至12章的3个项目（底层编程、高级编程和操作系统）则采用之前项目中实现的汇编语言和高级语言来编写。

项目提示 本书总共有12个项目。按照一般的大学课程安排，完成每个项目大约会占用一周课外作业时间。这些项目完全是各自独立的，学生可以按任何顺序来完成（或者选择其中的几个来完成）。当然，“完整的体验”则需要按照顺序来逐个实现，这只是其中一种选择而已。

在讲授这门课程的时候，我们会做两个大“让步”。首先，除非是一些关键的情况，否则我们不去考虑优化问题，这个重要课题还是留给相应的专业课程来涵盖。其次，在开发翻译器软件包（汇编编译器、VM实现和编译器）的时候，我们提供了无错误的测试文件（源程序），以便让学生在翻译器的输入没有错误的前提下对自己构建的模块进行测试。因此，学生不必编写处理错误和异常的代码，从而使得软件项目更易于管理。当然，处理错误输入是一门重要课程，但我们认为学生可以在其他课程中学习这些相关的内容，比如专门的编程和软件设计课程。

软件
Software

本书的支持网站（*www.idc.ac.il/tecs*）提供了用于构建书中描述的所有硬件和软件系统所需的工具和资源。其中包括硬件仿真器、CPU仿真器、VM仿真器和汇编编译器的可执行版本，虚拟机、编译器，以及本书所介绍的操作系统。网站还提供了所有项目需要的资源——大约两百个测试程序和测试脚本，用于各个项目的开发和单元测试。提供的所有软件工具和项目资源都可以在任何装有Windows或Linux操作系统的计算机上使用。

致谢
Acknowledgments

本书中的所有软件都是由Interdisciplinary Center Herzliya的Efi Arazi计算机科学学院的学生们开发的。首席软件架构师是Yaron Ukrainitz，开发人员包括Iftach Amit、Nir Rozen、Assaf Gad和Hadar Rosen-Sior。与这些学生兼开发人员一起工作非常愉快，能够在他们的教育生涯中担任他们的老师，我们深感荣幸和自豪。也要感谢我们的助教，Muawyah Akash、David Rabinowitz、Ran Navok和Yaron Ukraintz，他们帮助我们针对这门课程作了很多前

期探索性的教学（这正是本书的基础蓝本）。还要感谢 Jonathan Gross 和 Oren Baranes，他们在 Danny Seidner 博士的精心指导下从事着相关项目的研究；感谢 Uri Zeira 和 Oren Cohen，他们为 Jack 语言设计了集成开发环境；感谢 Tal Achituv 在开源问题方面的建议；感谢 Aryeh Schnall 仔细阅读本书并提出了一些编辑方面的细致建议。

在编写本书的同时还要兼顾我们日常的教学任务是不太容易的，所以我们还要感谢 Efi Arazi 计算机科学学院的主任 Esti Romen 在困难时期为我们承担了很多责任。最后，我们要感谢那些使用过本书早期版本并帮助我们改正了很多错误的许多学生。希望他们在学习过程中领会到 James Joyce（詹姆斯 •乔伊斯，著名作家）所洞察的真谛："mistakes are the portals of discovery（错误是发现探索的温床）"。

Noam Nisan

Shimon Schocken

目录

Contents

前言 xv

介绍：Hello, World Below 1

第 1 章　布尔逻辑 7

- 1.1　背景知识 8
  - 1.1.1　布尔代数 8
  - 1.1.2　门逻辑 11
  - 1.1.3　实际硬件结构 13
  - 1.1.4　硬件描述语言(**HDL**) 14
  - 1.1.5　硬件仿真 17
- 1.2　规范详述 17
  - 1.2.1　Nand 门 19
  - 1.2.2　基本逻辑门 19
  - 1.2.3　多位基本门 21
  - 1.2.4　多通道逻辑门 23
- 1.3　实现 25
- 1.4　观点 26
- 1.5　项目 27

第 2 章　布尔运算 29

- 2.1　背景知识 30
- 2.2　规范详述 32
  - 2.2.1　加法器 32
  - 2.2.2　算术逻辑单元(ALU) 35
- 2.3　实现 38
- 2.4　观点 39
- 2.5　项目 40

第 3 章 时序逻辑 41
3.1 背景知识 42
3.2 规范详述 47
3.2.1 D 触发器 47
3.2.2 寄存器 48
3.2.3 存储 49
3.2.4 计数器 50
3.3 实现 50
3.4 观点 52
3.5 项目 54
第 4 章 机器语言 57
4.1 背景知识 58
4.1.1 机器 58
4.1.2 语言 59
4.1.3 命令 60
4.2 Hack 机器语言规范详述 62
4.2.1 概述 62
4.2.2 A-指令 64
4.2.3 C-指令 66
4.2.4 符号 69
4.2.5 输入/输出处理 70
4.2.6 语法规约和文件格式 71
4.3 观点 72
4.4 项目 73
第 5 章 计算机体系结构 79
5.1 背景知识 80
5.1.1 存储程序概念 80
5.1.2 冯·诺依曼结构 80
5.1.3 内存 81
5.1.4 中央处理器 82
5.1.5 寄存器 83
5.1.6 输入和输出 84

5.2 Hack 硬件平台规范详述 85
5.2.1 概述 85
5.2.2 中央处理器（CPU） 87
5.2.3 指令内存 87
5.2.4 数据内存 87
5.2.5 计算机 91
5.3 实现 91
5.3.1 中央处理器 93
5.3.2 内存 96
5.3.3 计算机 96
5.4 观点 96
5.5 项目 99

第 6 章 汇编编译器 103

6.1 背景知识 104
6.2 Hack 汇编到二进制的翻译规范详述 107
6.2.1 语法规约和文件格式 107
6.2.2 指令 108
6.2.3 符号 110
6.2.4 范例 111
6.3 实现 112
6.3.1 Parser 模块 112
6.3.2 Code 模块 114
6.3.3 无符号程序的汇编编译器 114
6.3.4 SymbolTable 模块 115
6.3.5 有符号程序的汇编编译器 115
6.4 观点 117
6.5 项目 117

第 7 章 虚拟机 I：堆栈运算 121

7.1 背景知识 122
7.1.1 虚拟机范型 122
7.1.2 堆栈机模型 124
7.2 VM 规范详述，第 I 部分 127
7.2.1 概论 127

7.2.2 算术命令和逻辑命令 130
7.2.3 内存访问命令 130
7.2.4 程序流程控制命令和函数调用命令 133
7.2.5 Jack-VM-Hack 平台中的程序元素 133
7.2.6 VM 编程实例 135
7.3 实现 139
7.3.1 Hack 平台上的标准 VM 映射，第 I 部分 139
7.3.2 关于 VM 设计实现的建议 143
7.3.3 程序结构 144
7.4 观点 146
7.5 项目 147

第 8 章 虚拟机 II：程序控制 153

8.1 背景知识 153
8.1.1 程序控制流 155
8.1.2 子程序调用 155
8.2 VM 规范详述，第 II 部分 159
8.2.1 程序控制流命令 159
8.2.2 函数调用命令 159
8.2.3 函数调用协议 160
8.2.4 初始化 161
8.3 实现 161
8.3.1 Hack 平台上的标准 VM 映射，第 II 部分 161
8.3.2 范例 165
8.3.3 VM 实现的设计建议 168
8.4 观点 169
8.5 项目 170

第 9 章 高级语言 173

9.1 背景知识 174
9.1.1 范例 1：Hello World 174
9.1.2 范例 2：过程化编程和数组处理 175
9.1.3 范例 3：抽象数据类型 175
9.1.4 范例 4：链表实现 179
9.2 Jack 语言规范详述 181

9.2.1 语法要素 181
9.2.2 程序结构 181
9.2.3 变量 183
9.2.4 语句 187
9.2.5 表达式 187
9.2.6 子程序调用 188
9.2.7 Jack 标准库 190
9.3 编写 Jack 应用程序 193
9.4 观点 195
9.5 项目 196

第 10 章 编译器 I：语法分析 199

10.1 背景知识 200
10.1.1 词法分析 202
10.1.2 语法 203
10.1.3 语法分析（Parsing） 203
10.2 规范详述 207
10.2.1 Jack 语言语法 207
10.2.2 Jack 语言的语法分析器 207
10.2.3 语法分析器的输入 210
10.2.4 语法分析器的输出 210
10.3 实现 213
10.3.1 JackAnalyzer 模块 213
10.3.2 JackTokenizer 模块 214
10.3.3 CompilationEngine 模块 215
10.4 观点 217
10.5 项目 218

第 11 章 编译器 II：代码生成 223

11.1 背景知识 224
11.1.1 数据翻译 224
11.1.2 命令翻译 231
11.2 规范详述 232
11.2.1 虚拟机平台之上的标准映射 233
11.2.2 编译过程举例 235

11.3 实现 237
11.3.1 JackCompiler 模块 238
11.3.2 JackTokenizer 模块 238
11.3.3 SymbolTabel 模块 238
11.3.4 VMWriter 模块 240
11.3.5 CompilationEngine 模块 241
11.4 观点 241
11.5 项目 242
第 12 章 操作系统 247
12.1 背景知识 248
12.1.1 数学操作 248
12.1.2 数字的字符串表示 252
12.1.3 内存管理 252
12.1.4 变长数组和字符串 256
12.1.5 输入/输出管理 256
12.2 Jack OS 规范详述 263
12.2.1 Math 264
12.2.2 String 264
12.2.3 Array 265
12.2.4 Output 265
12.2.5 Screen 265
12.2.6 Keyboard 266
12.2.7 Memory 266
12.2.8 Sys 267
12.3 实现 267
12.3.1 Math 268
12.3.2 String 268
12.3.3 Array 269
12.3.4 Output 269
12.3.5 Screen 269
12.3.6 Keyboard 270
12.3.7 Memory 270
12.3.8 Sys 271

12.4 观点 272
12.5 项目 273

第 13 章 后记：发掘更多乐趣 277

13.1 硬件的实现 278
13.2 硬件的改进 278
13.3 高级语言 279
13.4 优化 279
13.5 通信 279

附录 A： 硬件描述语言（HDL） 281

A.1 范例 281
A.2 约定 282
A.3 将芯片加载到硬件仿真器 283
A.4 芯片描述头（接口） 284
A.5 芯片描述体（实现） 284
A.5.1 单元 284
A.5.2 管脚和连接 285
A.5.3 总线 286
A.6 内置芯片 287
A.7 时序芯片 289
A.7.1 时钟 289
A.7.2 时钟芯片和管脚 290
A.7.3 反馈环 291
A.8 芯片操作的可视化 292
A.9 已经提供的内置芯片与新的内置芯片 293

附录 B： 测试脚本语言 297

B.1 文件的格式和用法 298
B.2 在硬件仿真器中测试芯片 299
B.2.1 范例 299
B.2.2 数据类型和变量 299
B.2.3 脚本命令 301
B.2.4 内置芯片的方法和变量 304
B.2.5 范例 305

B.2.6 默认脚本 306
B.3 在 CPU 仿真器中测试机器语言程序 306
B.3.1 范例 308
B.3.2 变量 309
B.3.3 命令 309
B.3.4 默认脚本 309
B.4 在 VM 仿真器中测试 VM 程序 310
B.4.1 范例 311
B.4.2 变量 311
B.4.3 命令 313
B.4.4 默认脚本 313

索引 315

介绍：Hello, World Below

The true voyage of discovery consists not of going to new places, but of having a new pair of eyes.

真正的发现之旅不在于去新的地方，而在于拥有一双新的眼睛。

——Marcel Proust（1871 ~ 1922），作家

本书带你踏上发现之旅。你将学到三种知识：一、计算机如何工作；二、如何将复杂问题分解为易于管理的模块；三、如何开发大规模硬件和软件系统。整个学习过程是实践的过程，你将会从零开始创建一个完整的、可工作的计算机系统。在进行实践的同时，你也会学到一些远比计算机本身更加重要的知识。心理学家 Carl Rogers 曾说过："唯一能够显著影响行为的学习就是自我发现或自我适应——真理汲取自经历体验。"本章为你勾勒出本书中的发现、真理和经历体验。

上面的世界
The World Above

如果你学习过某门程序设计课程，在刚开始学习的时候可能遇到了类似于如下程序的内容。该程序是用 Jack（一门简单的、具有典型的基于对象的高级语言语法）编写的。

```
// 程序设计课程中的第 1 个例子:
class Main {
   function void main() {
     do Output.printString("Hello World");
     do Output.println();  // 换行.
     return;
   }
}
```

像 Hello World 这样的小程序表面上看起来十分简单。你可曾想过如何才能在计算机中运行这个程序呢？让我们来揭示其中的原委。对于初学者而言，程序只是一堆存储在文本文件中的字符而已。因此我们必须要做的第一件事情就是对该文本进行语法分析，揭示其语义，然后用某种计算机能理解的低级语言来重新表达程序。这个翻译过程（即**编译**，compilation）产生的结果就是生成另外一个包含机器代码的文本文件。

当然，机器语言也是一种抽象，是一组计算机能理解的二进制代码。为了将该抽象形式具体化，就必须由某种**硬件体系**（*hardware architecture*）来实现。这个硬件体系又是由**芯片组**（*chip set*）——寄存器、内存单元、ALU 等——来实现的。其中每个硬件设备都是由许多**基本逻辑门**（*logic gates*）集成构建出来的。而这些门又是由诸如 *Nand* 和 *Nor* 这样的**原始门**（*primitive gates*）构建而成的。当然，每个原始门又是由若干个**转换设备**（*switching devices*）组成，这些设备一般用晶体管实现。每个晶体管又是由……好了，我们不再继续深究下去，因为那已经到达了计算机科学领域的尽头，物理领域的开端。

你可能会想："在**我的**计算机上，编译和运行程序是相当容易的——我只要点一些按钮或编写一些命令！"确实，现代计算机系统就像一座巨大的冰山，大多数人只看到了最顶端的一角。大多数人掌握的计算系统知识也是粗略和浅显的。如果你希望能够探索到底层去，那你反而是幸运的！在那个精彩的世界，具有计算机科学领域里面的一些最美丽的东西。对底层世界的深刻理解也是区分普通程序员和高级开发者（既能开发应用程序，也能开发复杂的硬件和软件系统的人）的标准。理解（我们指的是深之入骨的透彻理解）这些系统工作原理和方式的最好方法就是从头开始构建完整的计算机系统。

抽象
Abstractions

你可能会觉得奇怪，如何能够仅利用一些基本的逻辑门来从零开始去构建完整的计算机系统。这一定是极为复杂的任务！解决这种复杂性的办法是，将整个项目划分为许多个**模块**（*modules*），然后在本书的每一章里单独处理其中一个模块。你可能还会觉得奇怪，如何可能单独地描述和构建这些模块呢？它们显然都是相互关联的呀！正如我们在书中将

要展示的，好的模块化设计意味着：可以单独处理每个模块，而完全不管系统的其他部分。实际上，你甚至可以按照任意顺序来构建这些模块！

这种良好的设计方式得益于人类独一无二的天赋：我们所拥有的建立和使用**抽象概念**（*abstractions*）的能力。一般所谓抽象概念，是作为表达思想的方式，将事物本质性的东西从思维上独立出来，以求用概括的方式来把握事物。而在计算机科学领域里，我们将抽象的概念定义得非常具体，认为它是关于“事物要做什么”的概念，而不用考虑“如何来做”。这种功能性定义必须包含足够的信息以便使用该事物能够提供的服务。事物在实现中的所有内容（包括技巧、内部信息、精妙之处等），都对要使用该事物的客户隐藏起来，因为这些与客户并没有关系。对于所要进行的专业实践而言，我们就需要建立、使用并实现这种抽象。硬件和软件开发者都会定义抽象（也称为“接口”，interfaces），然后进行实现或留给其他人来实现。抽象通常是分层构建（一层构建在另一层之上），从而形成了越来越高层级的抽象能力。

设计良好的抽象是一门实践艺术，掌握这门艺术的最好方法就是学习很多例子。因此，本书基于“先抽象再实现”的范式来进行阐述。每一章都介绍了一个关键的硬件或软件抽象，以及实际实现该抽象的实践项目。抽象的模块化特性使得每一章的内容各自独立，读者只需关心两件事情：理解抽象，然后利用抽象服务和底层构建模块来实现它。在本书的发现之旅的途中前进时，每次你回头看，都会欣喜地发现计算机系统正在你的努力之下逐渐成形。

下面的世界
The World Below

计算系统设计中蕴含的多层（multi-tier）抽象结构可以用**自顶向下**（*top-down*）的形式来描述，以此来展示高级抽象如何被简化或表示成较简单抽象。同样的结构也可以用**自下而上**（*bottom-up*）的形式来描述，以此展示底层抽象如何构建更复杂的抽象。本书采用后一种方式进行阐述：从最基本的元素（原始逻辑门）开始，然后一直向上层进发，直到最后构建完整的计算机系统。如果说构建这样的系统好比攀登珠穆朗玛峰，那么让计算机

运行用某种高级语言编写的程序就好比是在峰顶插上一只旗子。由于我们准备从山底开始逐渐向上攀登，因此我们先从反方向的角度来概览本书结构，首先从大家最熟悉的高级程序设计开始。

概览主要有三个部分。从最顶端开始，人们能够编写和运行高级程序（第 9 章和第 12 章）。然后开始探索通向硬件领域的道路，去领略将高级程序翻译成低级语言过程中的盘山曲径（第 6、7、8、10、11 章）。最后到达最底层，描述如何构建典型的硬件平台 （第 1 至 5 章）。

高级语言的领地
High-Level Language Land

我们的旅程中最高级的抽象就是程序设计艺术，企业家和程序员们构想实际应用，然后程序员们编写代码来实现这些构想。他们的工作中有两个关键的工具，在他们看来是理所当然就具备了的：1）所使用的高级语言；2）支持高级语言的丰富的服务程序库。例如，考虑语句 `do Output.printString("Hello World")`。这条代码调用了用于打印字符串的抽象服务，而该服务必须是在某个地方已经被实现了才行。继续深究不难发现，这个打印字符串服务通常是由操作系统和标准语言程序库联合提供的。

那什么是**标准语言程序库**（*standard language library*）呢？**操作系统**（OS，*Operating System*）又是如何工作的呢？这些问题会在第 12 章予以讨论。第 12 章首先介绍一些与 OS 服务相关的关键算法，然后利用这些算法来实现各种数学函数、字符串操作、内存分配任务和输入/输出（I/O）程序。最后得到的就是用 Jack 语言编写的简单操作系统。

Jack 是一门简单的基于对象（object-based）的语言，仅用来说明现代编程语言（如 Java 和 C#）的设计和实现中所蕴含的关键的软件工程思想。第 9 章中详细介绍 Jack 语言，并说明如何构建 Jack 应用程序（比如小游戏）。若你具有面向对象语言的编程经验，则马上就能编写 Jack 程序，并在本书第 9 章之前开发的计算机平台上观察其运行情况。但是，

第 9 章的目的不是让你成为 Jack 程序员，而是为后续章节中开发编译器和操作系统作必要准备。

向下通往硬件领地之路

The Road Down to Hardware Land

任何程序在实际运行之前，首先必须被翻译成某种目标计算机平台的机器语言。这个**编译**（*compilation*）过程十分复杂，于是被划分为若干个抽象层级，这些抽象层一般包含三种翻译器：一是编译器，二是虚拟机，三是汇编编译器。我们会用 5 个章节的篇幅来介绍这三个内容。

我们将**编译器**（*compiler*）的任务从概念上分为两个阶段：**语法分析**（*syntax analysis*）和**代码生成**（*code generation*）。首先在第一阶段，要对源文本进行分析，然后将其分组成有意义的语言结构，并将这些结构保存在称为“语法分析树（parse tree）”的数据结构中。这种进行语法分析的任务，统称为**语法分析**（*syntax analysis*），会在第 10 章中进行阐述。这为第二阶段作好了准备。第 11 章阐述语法分析树如何被递归处理，以便生成用中间语言编写的程序。Jack 编译器生成的中间代码描述了在基于堆栈的虚拟机（VM）上的一系列通用的操作步骤。第 7 至 8 章介绍这种模型以及实现在其上的虚拟机。由于 VM 的输出是一个大的汇编程序，因此必须将其进一步翻译成二进制代码。编写汇编编译器是相对容易的任务，会在第 6 章中进行阐述。

硬件的领地

Hardware Land

现在我们已经从机器语言到机器本身，到达旅程中最具奥秘的终点，软件终于在此处与硬件会合。*Hack* 也就在这里进入到我们眼前的图景当中。Hack 是一个通用计算机系统，力求在简单性和功能性之间取得平衡。利用第 1 至 3 章中介绍的芯片组和相关指导信息，我们可以在几个小时之内就构建好 Hack 体系结构。同时，Hack 也具有足够的通用特性，足以用来说明任何数字计算机设计中的关键操作原则和硬件元素。

第 4 章介绍了 Hack 平台的机器语言，而计算机设计本身则在第 5 章中讨论。利用与本书配套的基于软件的硬件仿真器以及附录 A 中介绍的硬件描述语言（HDL，Hardware Description Language），读者可以在自己的计算机上构建这个基于 Hack 平台的计算机以及本书所提到的所有芯片和门电路。所有开发出来的硬件模块都可以利用本书提供的测试脚本来进行测试（关于测试脚本语言可以参考附录 B）。

由此构建而成的计算机是基于典型的设备的，比如 CPU、RAM、ROM、模拟屏幕和键盘等组件。在第 3 章中简单地讨论时序逻辑并构建计算机的寄存器（registers）和内存系统（memory systems）。在第 2 章中简单地介绍布尔运算，然后构建计算机的组合逻辑（最终完成算术逻辑单元即 ALU 芯片的构建）。在这些章节中涉及的所有芯片都是基于一组基本逻辑门而构成，我们在第 1 章介绍并构建这些基本逻辑门。

当然，抽象层级并不止于此。基本逻辑门是利用基于固态物理理论以及量子理论的技术，由晶体管构建而成的。事实上，正是在这里，**自然世界**（*natural world*）之抽象（由物理学家来研究和描述）才演化成为**人造世界**（*synthetic world*）之抽象（由计算机科学家构建和研究）的构建模块。

到这里我们就结束了整个旅程概览——从基于对象的软件的最高层级一直下降到了构建硬件平台的砖瓦泥土。这种典型的多层次系统的模块化结构不仅体现了一种功能强大的工程范式，也反映了人为推理中的中心教条之一，这可以追溯到 2 500 年前：

> 我们所考虑的不是目的，而是朝向目的的实现的东西。医生并不考虑是否要使一个人健康，演说家并不考虑是否要去说服听众……他们是先确定一个目的，然后来考虑用什么手段和方式来达到目的。如果有几种手段，他们考虑的就是哪种手段最能实现目的。如果只有一种手段，他们考虑的就是怎样利用这一手段去达到目的，这一手段又需要通过哪种手段来获得。这样，他们就在所发现的东西中一直追溯到最初的东西……分析的终点也就是起点。（*Aristotles, Nicomachean Ethics, Book III, 3, 1112b*[1]）

本书的计划就是按照实现的顺序来展开阐述。首先是基本逻辑门的构建（第 1 章），然后由下而上的构建组合芯片和时序芯片（第 2 至 3 章），从典型的计算机体系架构（第 4 至 5 章）和软件层级（第 6 至 8 章），一直讲到实现基于对象的语言（第 9 章）的编译器（第 10 至 11 章），最终实现一个简单的操作系统（第 12 章）。希望读者已经对这些任务有了基本概念，并渴望开始这段冒险之旅。好的，假设你准备好了，让我们开始倒数：1，0，出发！

---

[1] 亚里士多德《尼各马可伦理学》第三卷第 3 节 1112b。本中文译文摘自汉译世界学术名著丛书《尼各马可伦理学》第三卷第 3 节 1112b 的第 10 行至第 24 行（即该书第 68 至 69 页），亚里士多德 著，廖申白 译注，商务印书馆，2003 年 11 月第一版。此书对“分析的终点也就是起点”一句有译注：亚里士多德此处的意思似乎是指完成一种考虑的起点。考虑就是考虑做事情要从哪里着手。所以找到了这个点（起点），考虑便完成，行动便开始。——审校者

第 1 章　布尔逻辑

Boolean Logic

Such simple things, And we make of them something so complex it defeats us, Almost.

一些简单的事情，我们往往把它们搞得很复杂以至于几乎使我们失败。

——John Ashbery（1927），美国诗人

每种数字设备——比如个人电脑、手机或者网络路由器——都是基于一组用于储存和处理信息的芯片构建而成的。虽然这些芯片在外形上、构成上不尽相同，但是它们都是由相同的构造模块制造而成的：基本的**逻辑门**（*logic gates*）。这些逻辑门可能会由很多不同的原材料和制造工艺来实现，但是其逻辑行为在所有的计算机中都是一致的。在本章中，我们从原始的逻辑门“与非门”出发，以此来构建其他逻辑门。得到的是一组相当标准的逻辑门，这些逻辑门在后面将会被用来构建计算机的处理和存储芯片，具体内容分别会在第 2 章和第 3 章介绍。

从本章开始，本书中所有的硬件相关章节都具有相同的内容组织结构。每一章都会集中讨论一个明确的设计任务——构建或整合某一组芯片。每一章开头的**背景知识**部分都会对掌握该设计任务所需要的预备知识作简要的背景介绍。接下来的**规范详述**部分会对芯片的逻辑抽象作详述，即这些芯片通过某种方式所能实现的各种功能。在了解了这些功能后，再接下来的**实现**部分将会对我们如何真正实现这些芯片提供思路和线索。在**观点**这一节中，将对该章遗漏的一些重要问题作总结。每章的最后都有一个技术**项目**。这一部分会给大家在个人计算机上实际构建芯片提供具体到每一操作步骤的指导，实验中将会用到随本书提供的硬件仿真器。

因为本章是全书第一个硬件章节，**背景知识**部分的介绍相对较长，在这一节里面特别介绍了**硬件描述**（*hardware description*）和**仿真工具**（*simulation tools*）。

1.1 背景知识 Background

本节主要介绍一组简单的芯片即**布尔门**（*Boolean gate*）的构建。因为布尔门是对**布尔函数**（*Boolean function*）的物理实现，因此我们首先对布尔代数（Boolean algebra）作简单的介绍。然后我们介绍如何将这些简单的布尔门相互连接来实现复杂芯片的功能。本节的末尾介绍了如何在实际中使用软件仿真来进行硬件设计。

1.1.1 布尔代数 Boolean Algebra

布尔代数处理布尔型（也称为二进制型）数值，比较典型的有 true/false、1/0、yes/no、on/off 等等。在这里我们使用 1 和 0。布尔型函数（Boolean function）是指输入输出数值均为布尔型数值的函数。因为计算机硬件基于二进制数据的表示和处理，所以布尔函数在硬件体系结构的描述、构建和优化过程中扮演着十分重要的角色。那么正确地表达和分析布尔函数是迈向构建计算机体系结构的第一步。

真值表表示法（Truth Table Representation）　描述布尔函数最简单的方法就是枚举出函数所有可能的输入变量组合，然后写出每一种组合所对应的函数输出值。这就是真值表表示法，参看图 1.1。

图 1.1 中的前三列枚举出了函数变量所有的二进制值，有 2^n 个可能的元组 $v_1 \ldots v_n$（这里 $n=3$），对于任何一个元组，最后一列给出对应的值 $f(v_1 \ldots v_n)$。

布尔表达式（Boolean Expressions）　除了真值表描述之外，布尔函数还可以在输入变量上使用布尔算子（Boolean operator）来描述。基本的布尔算子有 “And”（只有在 x 和 y 都为 1 时，x And y 的值等于 1），“Or”（x 或者 y 中只要有一个为 1 时，x Or y 的值为 1），以及 “Not”（当 x 为 0 时，Not x 为 1）。我们将使用常用的计算符号来表示这些算子：$x \cdot y$（或者 xy）代表 x And y，$x+y$ 代表 x Or y，$\overline{x}$ 代表 Not x。

我们再来看看图 1.1，该真值表可以用布尔表达式 $f(x,y,z)=(x+y)\cdot\overline{z}$ 来表示。

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

图 1.1　布尔函数真值表表示

例如，当输入 $x=0$，$y=1$，$z=0$（表中的第三行）时，$x+y=1$，因此$1\cdot\bar{0}=1\cdot1=1$。如果要完全验证布尔表达式和真值表的一致性，就取每一种可能的输入变量值的组合，看通过表达式计算出来的结果是否和表中最右列的值一致。

规范表示法（Canonical Representation）　每个布尔函数都至少由一个布尔表达式来描述，称之为**规范表示法**（*canonical representation*）。同样从函数的真值表出发，我们关注函数值为 1 的那些行。对于每一行，我们构建一种表达法来确定所有输入变量的值，即通过对两个字面量（literals）（每个量可以是变量本身或其反值）施以 And 操作来得到。比如说图 1.1 中的第三行，函数的值为 1。输入变量为 $x=0$，$y=1$，$z=0$，于是我们构建表达式 $\bar{x}y\bar{z}$。同样地，$x\bar{y}\bar{z}$ 和 $xy\bar{z}$ 分别代表第 5 行和第 7 行。现在如果我们把这些表达式用 Or 操作联合起来（对于所有的函数值为 1 的行），那么可以得到跟给定的真值表一致的布尔表达式。于是图 1.1 中的布尔函数的规范表示法可以写成 $f(x, y, z) = \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z}$。由这种表示法得出一个重要结论：每个布尔函数，不管有多复杂，都可以只使用三个布尔算子 And、Or、Not 来完全表达。

2-输入变量的布尔函数（Two-Input Boolean Functions）　图 1.1 揭示了 n 个二进制变量能够构成的布尔函数的数量为 2^{2^n}。例如，图 1.2 中列出了 2 个输入变量所能构成的 16 个布尔函数。右边的四列枚举出所有可能的二进制值的组合，每个函数都有常规的名字来描述它蕴含的操作。这里举了一些例子：名称叫 Nor 的函数实际上是 Not-Or 的简称：

先对 x 和 y 进行 Or 操作，然后取反。Xor 函数则是异或的简称，即当两个变量的值相异时则返回 1，相同时返回 0。相反，Equivalence 函数则是当变量取值相同时返回 1，相异时返回 0。If-x-then-y 函数（也称为 $x \rightarrow y$，或者“x implies y”）是当 x 为 0 或当 x 和 y 都为 1 时返回 1。表中其他函数就不再一一说明了。

Function	x	**0**	**0**	**1**	**1**
	y	**0**	**1**	**0**	**1**
Constant 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
x And Not y	$x \cdot \bar{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\bar{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x + y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	$\bar{y}$	1	0	1	0
If y then x	$x + \bar{y}$	1	0	1	1
Not x	$\bar{x}$	1	1	0	0
If x then y	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

图 1.2　所有“两变量”的布尔函数

Nand 函数以及 Nor 函数在理论上还有个有趣的特征：And/Or/Not 算子都可以只用 Nand 或 Nor 函数来建构，（比如，x Or y = (x Nand x) Nand (y Nand y)）。既然每个布尔函数都能够通过规范表示法由 And、Or 和 Not 构成，那么每个布尔函数也能仅使用 Nand 函数来构成。这个结论有比较深远的实际意义：一旦在物理上实现了 Nand 功能，就可以使用很多这样的物理设备，通过特定的连接方式来构建任何布尔函数的硬件实现。

1.1.2 门逻辑
Gate Logic

门（*gate*）是用来实现布尔函数的物理设备。如果布尔函数 f 有 n 个输入变量，返回 m 个二进制的结果（在我们前面所举的所有例子里面，$m = 1$），那么用来实现这个函数 f 的门将会有 n 个**输入管脚**（*input pins*）和 m 个**输出管脚**（*output pins*）。当我们把一些值 $v_1 \ldots v_n$ 从这些门的输入管脚输入，它的内部结构即门的逻辑会计算然后输出 $f(v_1 \ldots v_n)$ 的值。好比复杂的布尔函数能够通过相对简单的函数来表达一样，复杂的门电路也是由很多基本的门组成的。最简单的门是由微小的开关设备（称为**晶体管**，transistors）构成，这些微小开关设备按照设计的拓扑结构进行连接，来实现整个门的功能。

虽然当今的计算机多数使用电学来表述二进制数据从一个门到另一个门的传递，但实际上任何具有转换（switching）和传导（conducting）能力的技术都是可用的。事实上，在过去的 50 年里，研究人员已经建立了很多布尔函数的硬件实现方法，包括磁、光、生物、水力和风力设备。今天，大多数门都采用晶体管来实现，这些晶体管在硅上蚀刻并封装成芯片。在本书里，我们所提到的**芯片**（*chip*）和门（*gate*）的概念是可交换使用的，但是门的概念一般用于描述简单的芯片。

有一点非常重要：一方面，我们有了多种可选的转换技术（switching technology），另一方面，我们又发现可以用布尔代数来对这些转换技术进行抽象。简而言之这就意味着：计算机科学家们不必担心物理上的细节，比如电流、电路、开关、延迟和电源等等。内部结构仅仅跟门电路设计者相关；而外部结构则是其他设计者所关心的问题——他们希望将门电路作为封装完好的抽象组件来使用，而不希望去考虑其内部结构。因此，原始的门电路（*primitive gate*，如图 1.3 所示）能够被看成黑箱子，应用这些黑箱子来实现逻辑操作；至于黑箱子内部的实现，我们并不关心。硬件设计人员从这些基本的门开始，将它们进行连接，来实现功能更复杂的复合（*composite*）门电路。

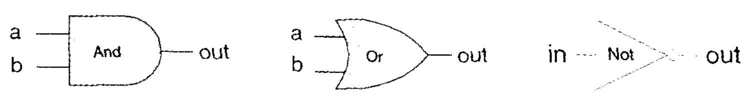


图 1.3 基本逻辑门的标准符号表示

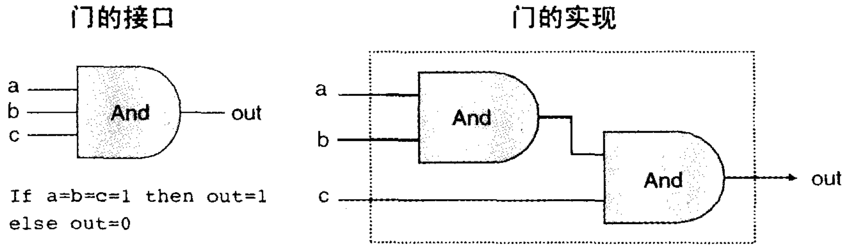


图 1.4 一个三通道"与"门的组合实现。右图中的矩形定义了门接口在概念上的边界

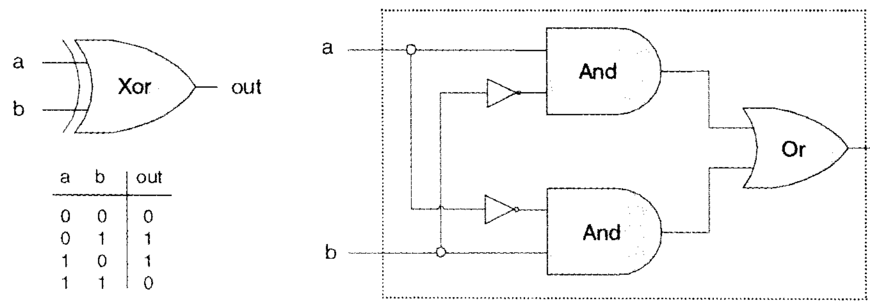
简单门和复合门（Primitive and Composite Gates） 所有的逻辑门都有相同输入和输出的语义表达（0 和 1），它们可以连起来构成具有任意复杂度的**复合门电路**（*composite gates*）。例如，要实现一个具有三个输入变量的布尔函数 And(a,b,c)，可以通过使用布尔代数，按照$a \cdot b \cdot c=(a \cdot b) \cdot c$或者前缀符号 And($a,b,c$)=And(And($a,b$),$c$)来实现。结果如图 1.4 所示。

图 1.4 描述的是一个简单**门逻辑**（*gate logic*）的例子，也称为**逻辑设计**（*logic design*）。有时候我们认为逻辑设计是一种连接门电路的艺术，目的是构建功能更复杂的门电路，即**复合门电路**。复合门电路本身就是某些布尔函数的实现，它们的"外部表现"（比如，图 1.4 中左边的图）跟基本门电路一致，虽然它们的内部结构可能会相当复杂。

对于任何给定的逻辑门，我们都能够从外部和内部两个不同方面对其进行观察。图 1.4 中右边的图给出了门的内部结构（或称为**内部实现**），而左边的部分仅仅显示了门的外部**接口**（*interface*），也就是输入和输出管脚。内部结构仅仅与门电路的设计者相关，而外部结构则是其他设计者所关心的，他们仅使用门电路的抽象而不去关心其内部结构。

我们来看看另一个逻辑设计的例子：Xor 门。如前面所讨论的一样，当 a 和 b 的取值相异的时候，Xor(a,b) 的值为 1。那么我们可以用这个式子来表示：Xor(a,b) =Or(And(a,Not(b)),And(Not(a),b))。通过这个表示法，我们可以得到如图 1.5 所示的逻辑设计。

必须注意门的**接口**是唯一的：一般通过真值表、布尔表达式或者一些字面表述来实现。



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

图 1.5 异或门及其可能的实现方案

但我们能够使用很多不同的方法来实现这个接口，其中一些方法要在开销、速度和简洁程度等方面占有优势。比如说，Xor 函数可以使用 4 个 And、Or、Not 门来实现，比 5 个门的实现方式要好。虽然**逻辑设计的基本功能可以通过这样或那样的方式实现其外部接口**，但从效率的角度上来说，基本原则是**用尽可能少的门来实现尽可能多的功能**。

总的来说，可以这样描述逻辑设计的艺术：给定门的描述（外部接口），通过应用已经实现的门，找到有效的方法来实现它。简单地说，这就是我们在本章后续内容中所要介绍的。

1.1.3 实际硬件结构

Actual Hardware Construction

在介绍了由简单门电路组成复杂门结构的逻辑之后，现在该讨论这些门是如何来构建复杂结构的。我们首先从一个简单的例子来开始讨论吧。

假设我们在自家的车库开了一家芯片加工店。第一个定单是制造一百个 Xor 门。我们用订单的订金去买了一支焊枪，一卷铜芯线和三只分别贴有“And 门”、“Or 门”、“Not 门”标签的箱子，每箱包含很多同类的基本门电路。每个门电路都像电源插头那样用塑料盒子封装起来，只有一些输入和输出管脚露在外面。开始工作之前，把图 1.5 钉在车库的墙上。首先，拿出两个 And 门，两个 Not 门和一个 Or 门，按照图中的布局把它们安装在电路板

上。然后，用铜线把这些门一一连接起来，用焊枪把线和各自的输入/输出管脚焊起来。如果我们仔细地按照图中所描述的电路进行连接，最后会留三个露在外面的线头，然后把线头和对应的三个管脚焊起来，把整个电路（除了三个管脚）封装在一个塑料盒子里，然后打上“Xor”的标记。制造好一个门之后，接下来就是重复地做上述工作。一天下来，我们把制造好的所有的芯片装在一只新的箱子里面，然后标上“Xor 门”。如果我们（或者其他人）以后要制造其他的芯片，可以使用这些 Xor 门作为基本的模块，就像我们以前使用 And 门、Or 门、Not 门一样。

读者们可能感到：在车库进行芯片制造会不会太落后，应该还有很多可以改进之处吧。也许给定的芯片图纸不能保证其正确性。虽然我们能够证明一些简单芯片（比如 Xor 门）的正确性，但是对于许多复杂的芯片来说我们就无能为力了。因此，我们必须进行经验性的测试：把芯片通上电源，分别给输入管脚注入高电平、低电平，通过不同的组合来验证芯片的输入是否和其描述一致。如果芯片的输出不是所期望的值，就得修正它的物理结构，这是一件相当麻烦的事情。即使我们验证出电路的设计是完全正确的，重复的芯片制造过程将会是耗时、容易产生错误的事情。一定有更好的办法吧？！

1.1.4 硬件描述语言(HDL)
Hardware Description Language (HDL)

今天，硬件设计者再也不会用他们的那双手来制造硬件了。取而代之的是，他们在计算机工作站上设计和优化芯片结构，使用结构化的建模形式比如**硬件描述语言**（*Hardware Description Language*），或 HDL（大家都知道 VHDL，其中 V 代表 *Virtual* 即虚拟的）。设计者通过编写 **HDL 程序**来描述芯片的结构，该程序将接受严格的测试。该测试是虚拟的，使用计算机进行仿真的虚拟测试：有个特殊的软件工具称为**硬件仿真器**（*hardware simulator*），它接受输入的 HDL 程序，在计算机内存中构建该虚拟芯片的内存映像。然后，设计者就可以让仿真器来测试芯片，通过输入变量的不同组合产生仿真的芯片的输出。将这些仿真的输出结果与预期的值进行比较，来验证我们的设计是否正确，是否满足客户的要求。

除了测试芯片的正确性之外，硬件设计者可能还比较关心一系列参数，比如计算速度、

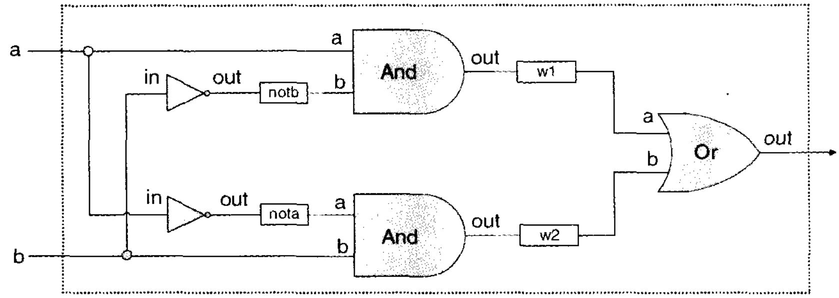
功耗和芯片设计的总成本等等。所有这些参数也可以利用硬件仿真器来仿真和量化，帮助设计者对设计进行优化直到仿真的芯片达到期望的性价比。

因此，在投资进行实际的芯片生产之前，可以用 HDL 对整个芯片进行全面的设计、调试、优化。当 HDL 程序完成之后，或者说，当仿真后的芯片性能满足客户的要求之后，HDL 程序才能被作为在硅上蚀刻芯片的设计图。芯片制造流程中的最后一步，即从优化的 HDL 程序到大量生产，是典型的外包过程（交给专注于芯片制造的公司来做），这其中要使用某种转换技术（switching technology）。

例子：构建 Xor 门　正如在图 1.2 和图 1.5 中看到的，描述异或逻辑的方式之一可以是 $\text{Xor}(a,b)=\text{Or}(\text{And}(a,\text{Not}(b)),\text{And}(\text{Not}(a),b))$ 。该逻辑同样可以用图形（如门结构图或文字，或 HDL 程序）来表达。Xor 的 HDL 表达法请参看图 1.6，关于 HDL 的介绍请参看附录 A。

解释　芯片的 HDL 定义包括 *header* 部分和 *parts* 部分。Header 部分描述了芯片的**接口**（*interface*），也就是芯片的名称、输入输出管脚的名称。parts 部分描述了所有底层电路的名称和拓扑结构，这些电路是构成该芯片的基本部分。每个部分用一个 *statement*（语句）来表示，它描述了该部分的名称以及与其他部分的连接方式。为了简单明了地编写这些语句，HDL 程序员必须有内在模块的**接口**文档。例如，图 1.6 中假定了 Not 门的输入输出管脚为 `in` 和 `out`，And 门和 Or 门的输入输出管脚分别为 `a`，`b` 和 `out`。这种类似 API 的信息并不会显式给出，但是将 parts 部分插入 HDL 代码之前，必须保证能够对这些信息进行访问才行。

内部模块的连接是通过建立和连接**内部管脚**（*internal pins*）来描述的。比如，图中 Not 门的输出管脚又和 And 门的输入管脚相连。HDL 代码通过 `Not(…,out=nota)` 和 `And(a=nota,…)` 来描述这个连接。第一条语句建立了一个名为 `nota` 的内部管脚，并作为 Not 门的输出。第二条语句又将 `nota` 作为 And 门的输入管脚。注意到管脚可能有无限的



***HDL* 程序** (Xor.hdl)	**测试脚本** (Xor.tst)	**输出文件** (Xor.out)
<pre>/* Xor (exclusive or) gate: If a<>b out=1 else out=0. */ CHIP Xor { IN a, b; OUT out; PARTS: Not(in=a, out=nota); Not(in=b, out=notb); And(a=a, b=notb, out=w1); And(a=nota, b=b, out=w2); Or(a=w1, b=w2, out=out); }</pre>	<pre>load Xor.hdl, output-list a, b, out; set a 0, set b 0, eval, output; set a 0, set b 1, eval, output; set a 1, set b 0, eval, output; set a 1, set b 1, eval, output;</pre>	<pre>a | b | out - - - - - - - - 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 0</pre>

图 1.6 异或门的 HDL 实现

扇出（fan out）。比如，在图 1.6 中，每个输入都同时被作为两个门的输入管脚。在门结构的图中，我们使用节点（forks）来表示多重连接。在 HDL 中，代码隐含了节点的存在。

测试 严格的质量认证要求设计出来的芯片必须在一种明确的、可重复和可读性强的方式下进行测试。应用这种思想，硬件仿真器通常被设计用来运行用脚本语言（scripting language）编写的**测试脚本**（*text script*）。例如，图 1.6 中的用脚本语言编写的测试脚本能够被硬件仿真器读懂。关于脚本语言的介绍请参看附录 B。

我们来对图 1.6 中介绍的测试脚本作简单地解释。测试脚本的前两行指示仿真器加载 Xor.hdl 文件并准备输出所选择的变量。 接着，脚本列出了一系列的测试场景

(testing scenarios)，用来模拟 Xor 芯片在真实环境中可能产生的各种结果。在每个测试场景中，脚本指示仿真器为芯片赋予确定的输入值，计算出对应的测试结果，然后将该结果记录到指定的输出文件中。对于一些简单的门（比如 Xor），可以用穷尽法编写完整的测试脚本来罗列出逻辑门电路所有可能的输入组合。输出文件（图 1.6 的右边）可以看作是凭经验得出的芯片的正确输出结果。然而，对于比较复杂的芯片，这种经验性的验证就不适用了，我们在后面也会看到这一点。

1.1.5 硬件仿真 Hardware Simulation

HDL 是一种**结构化的硬件语言**（*hardware construction language*），编写和调试 HDL 程序的过程跟软件开发相似。它同软件开发的主要区别是，它应用硬件仿真器，而不是在 HDL 中用 Java 语言编写代码或用编译器编译测试代码。硬件仿真器其实是个计算机程序，它知道怎样分析和解释 HDL 代码并且把它转换成可执行的表达式，然后根据给定的测试脚本来对代码进行测试。在市面上有很多商用硬件仿真器，它们在价格、复杂度和使用方便程度上各不相同。本书配套网站提供了一个简单的（免费的）硬件仿真器，对于一些成熟的硬件设计项目来说它是足够了。该仿真器提供了本书中所涉及的芯片设计、测试和集成过程中所需的全部工具，以此来构建通用的计算机系统。图 1.7 显示了一个典型的芯片仿真过程。

1.2 规范详述 Specification

这个部分详细介绍了一组典型的门电路，每个门被设计来执行一个常见的布尔操作。这些门将在接下来的一些章节里面被用到。我们从原始的 Nand 门出发，其他的门电路能够通过这个门电路构建得到。这里我们只提供门的说明或者接口，把具体的实现细节留在后续的章节中去介绍。读者如果有兴趣自己通过 HDL 来构建特定的门电路，可以参考附录 A。所有的门电路都能够在个人计算机上通过与本书配套的硬件仿真器来进行构建和仿真。

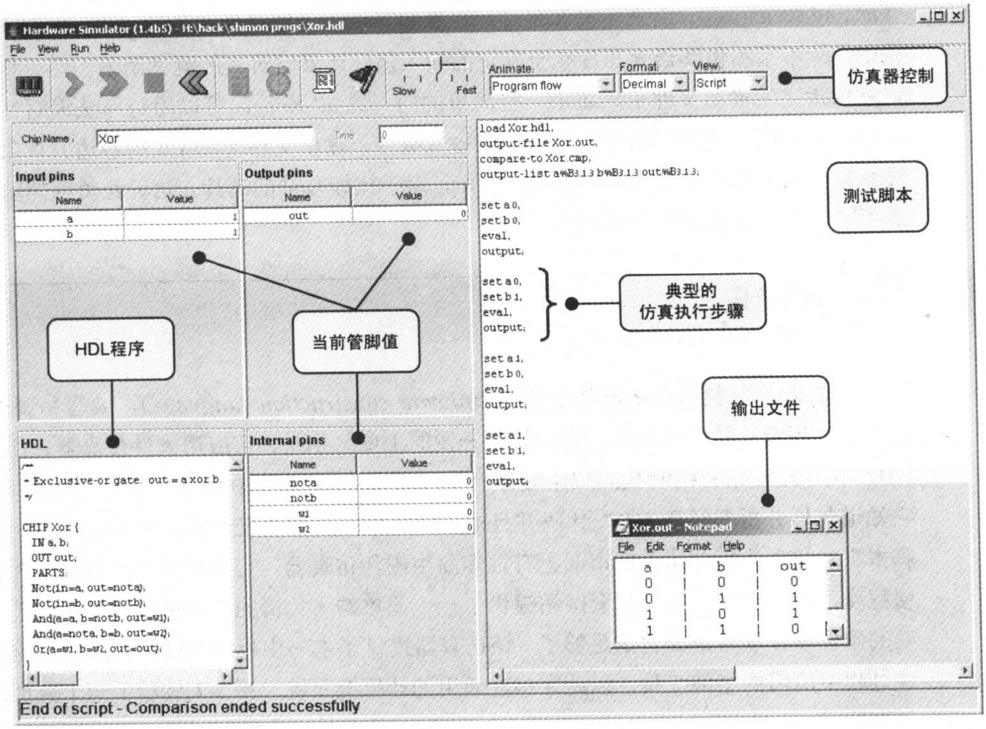


图 1.7　在硬件仿真器上对 Xor 芯片进行仿真的屏幕截图。图中显示的是测试脚本运行完毕后仿真器的状态。管脚的值与最后一个仿真步骤（$a = b = 1$）相对应。可以看出，仿真器生成的输出文件与 Xor 真值表是一致的，这也意味着被加载的 HDL 程序实现了正确的 Xor 功能。比较文件（*compare file*）并没有显示在图中，比较文件是由芯片的使用者指定的，与输出文件有相同的结构和内容。通过屏幕底部的状态栏的信息显示了两个文件的一致性

1.2.1 Nand 门
The Nand Gate

我们对计算机结构的讨论从简单的 Nand 门开始，所有其他的门电路和芯片能够通过它来构建。Nand 门用于实现以下布尔函数：

a	b	Nand(a, b)
0	0	1
0	1	1
1	0	1
1	1	0

纵观全书，我们使用“装箱的芯片 API”来描述芯片。对每个芯片，API 描述了芯片的名字、输入输出管脚的名字、芯片所能执行的功能或操作以及必要的说明。

```
芯片名:   Nand
输  入:   a, b
输  出:   out
功  能:   If a=b=1 then out=0 else out=1
说  明:   此门是最基本的单元，不需要实现。
```

1.2.2 基本逻辑门
Basic Logic Gates

这里提到的逻辑门都是“基本的”门电路。其中，每个门电路可仅由 Nand 门来组合构成。因此，它们不应该被看作是“原始”的门。

Not　单输入变量的 Not 门，也被称为“反相器（converter)”，将 0 反相为 1 输出或者将 1 反相为 0 输出。它的 API 如下：

```
芯片名:   Not
输  入:   in
输  出:   out
功  能:   If in=0 then out=1 else out=0.
```

And 只有当输入都是 1 时，And 函数输出 1，否则就输出 0。

```
芯片名:    And
输  入:    a, b
输  出:    out
功  能:    If a=b=1 then out=1 else out=0.
```

Or 只要输入变量中有 1，Or 函数就输出 1，否则输出 0。

```
芯片名:    Or
输  入:    a, b
输  出:    out
功  能:    If a=b=0 then out=0 else out=1.
```

Xor 又称为“异或”，当两个输入值相反的时候，Xor 函数输出 1，其他情况返回 0。

```
芯片名:    Xor
输  入:    a, b
输  出:    out
功  能:    If a=/b then out=1 else out=0.
```

Multiplexor Multiplexor（见图 1.8）是三输入变量的门电路，其中一个输入称为“选择位（Selection-bit）”，选择另外两个输入变量（称为数据位，data bits）中的一个作为输出，另外两个输入称为“数据位”。因此，这个门电路还有另一个比较好的名字：“选择器（selector）”。Multiplexor 这个名词来自于通信系统，意思是将多个相同设备的输出信号通过单一输出信号线串行地输出。

```
芯片名:    Mux
输  入:    a, b, sel
输  出:    out
功  能:    If sel=0 then out=a else out=b.
```

a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

sel	out
0	a
1	b

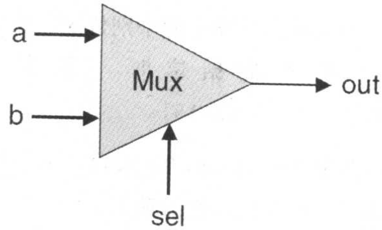


图 1.8 Multiplexor，右上角的表是左边真值表的简化版本

sel	a	b
0	in	0
1	0	in

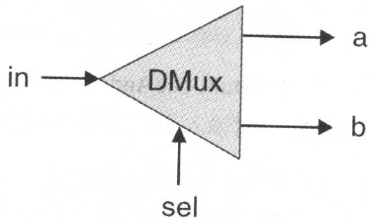


图 1.9 Demultiplexor

Demultiplexor Demultiplexor（见图 1.9）执行与 multiplexor 相反的功能：它只有一个输入变量，然后根据一个选择位来决定到底从哪一个通道输出。

芯片名：	DMux
输　入：	in, sel
输　出：	a, b
功　能：	If sel=0 then {a=in, b=0} else {a=0, b=in}.

1.2.3 多位基本门

Multi-Bit Versions of Basic Gates

通用计算机的设计要求其能够在多位数据线（又称为“总线”，bus）上运行。比如，32 位计算机的基本要求是能够在 2 个给定的 32 位总线上按位（bit-wise）进行 And 函数的计算，我们可以构建一个 32 位 And 门阵列，每个 And 门能够独立地处理一对输入变量的

And 运算。为了把所有的逻辑封装在同一个包里面，我们把这些门阵列压缩在一个单一的芯片接口里，包含两个 32 位的输入总线和一个 32 位的输出总线。

这一节介绍了一组典型的多位逻辑门，这些门在构建 16 位计算机时会用到。同 *n* 位逻辑门的结构是基本相同的，并与 *n* 的具体值无关。

当涉及到总线上的个别位时，我们通常使用数组的语法。比如说，在描述 16 位总线 `data` 的各数据位时，我们使用这样的表达法：`data[0]`, `data[1]`, `data[2]`,…,`data[15]`。

多位 Not（Multi-Bit Not） *n* 位的 Not 门所执行的布尔操作是：对它的 *n* 位输入总线上的每一位取反，然后输出。

```
芯片名:   Not16
输  入:   in[16] // 16-bit 管脚
输  出:   out[16]
功  能:   For i=0..15 out[i]=Not(in[i]).
```

多位 And（Multi-Bit And） *n* 位的 And 门所执行的布尔操作是：对两个 *n* 位输入总线上对应的每一对输入变量进行“与操作”，然后输出。

```
芯片名:   And16
输  入:   a[16], b[16]
输  出:   out[16]
功  能:   For i=0..15 out[i]=And(a[i],b[i]).
```

多位 Or（Multi-Bit Or） *n* 位的 Or 门所执行的布尔操作是：对两个 *n* 位输入总线上对应的每一对输入变量进行“或操作”，然后输出。

```
芯片名:   Or16
输  入:   a[16], b[16]
输  出:   out[16]
功  能:   For i=0..15 out[i]=Or(a[i],b[i]).
```

多位 Multiplexor（Multi-Bit Multiplexor） *n* 位 multiplexor 的结构跟图 1.8 中所描述的二进制 multiplexor 几乎完全一样，只是原来两个 1 bit 位的输入变量都变成了两个 *n* 位的输入变量；选择位仍然是 1 位。

```
芯片名:    Mux16
输  入:    a[16], b[16], sel
输  出:    out[16]
功  能:    If sel=0 then for i=0..15 out[i]=a[i]
           else for i=0..15 out[i]=b[i].
```

1.2.4 多通道逻辑门
Multi-Way Versions of Basic Gates

很多 2 位（即接收两个输入）的逻辑门能够推广到多位（即接收任意数量的输入）。这一节介绍一组多位门电路，它们将在以后介绍的计算机体系结构中的各种芯片中使用。相同的推广方式可以应用到其他的体系结构中。

多通道 Or（Multi-Way Or） 对于一个 *n* 位的 Or 门，当 *n* 位输入变量中任意一位或一位以上为 1，输出就为 1，否则就为 0。这里给出 8 位的例子。

```
芯片名:    Or8Way
输  入:    in[8]
输  出:    out
功  能:    out=Or(in[0],in[1],...,in[7]).
```

多通道/多位 Multiplexor（Multi-Way/Multi-Bit Multiplexor） 一个拥有 *m* 个通道、每个通道数据宽度为 *n* 位的 multiplexor 选择器，将 *m* 个 *n* 位输入变量中选择一个并从其单一的 *n* 位输出总线上输出。我们用 *k* 个控制位来指定这个选择，这里 $k=\log_2 m$。图 1.10 描述了一个典型的例子。

下面给出了两个 API 的例子：一个 4 通道 16 位的 mutiplexor 和一个 8 通道 16 位的 multiplexor。

sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

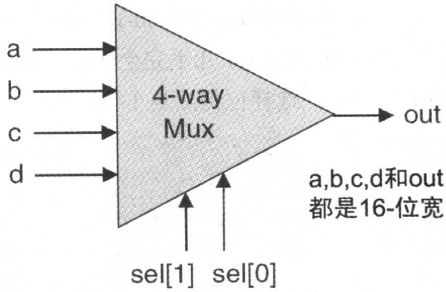


图 1.10 4-通道 multiplexor。输入总线和输出总线的宽度可以不同

```
芯片名:    Mux4Way16
输  入:    a[16], b[16], c[16], d[16], sel[2]
输  出:    out[16]
功  能:    If sel=00 then out=a else if sel=01 then out=b
           else if sel=10 then out=c else if sel=11 then out=d
说  明:    上述赋值操作都是 16-位操作。比如, "out=a" 意指 "for i=0..15 out[i]=a[i]"。
```

```
芯片名:    Mux8Way16
输  入:    a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16],
           sel[3]
输  出:    out[16]
功  能:    If sel=000 then out=a else if sel=001 then out=b
           else if sel=010 out=c ... else if sel=111 then out=h
说  明:    上述赋值操作都是 16-位操作。比如, "out=a" 意指 "for i=0..15 out[i]=a[i]"。
```

多通道/多位 Demultiplexor（Multi-Way/Multi-Bit Demultiplexor） m 通道、n 位的 demultiplexor（见图 1.11）从 m 个可能的 n 位输出通道中选择输出一个 n 位的输入变量。我们用 k 个控制位来指定这个选择，这里 $k=\log_2 m$。

下面给出了两个 API 的例子：一个 4 通道 1 位的 demultiplexor 和一个 8 通道 1 位的 demultiplexor。

sel[1]	sel[0]	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

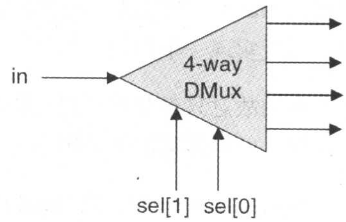


图 1.11 4-通道 demultiplexor

```
芯片名:   DMux4Way
输  入:   in, sel[2]
输  出:   a, b, c, d
功  能:   If sel=00 then {a=in, b=c=d=0}
          else if sel=01 then {b=in, a=c=d=0}
          else if sel=10 then {c=in, a=b=d=0}
          else if sel=11 then {d=in, a=b=c=0}.
```

```
芯片名:   DMux8Way
输  入:   in, sel[3]
输  出:   a, b, c, d, e, f, g, h
功  能:   If sel=000 then {a=in, b=c=d=e=f=g=h=0}
          else if sel=001 then {b=in, a=c=d=e=f=g=h=0}
          else if sel=010 ...
          ...
          else if sel=111 then {h=in, a=b=c=d=e=f=g=0}.
```

1.3 实现

Implementation

跟数学中**公理**的角色一样，**原始门电路**提供了一系列的基本模块，其他的门电路可以通过这些模块来构建。原始门是成熟的商用组件，因此可以被用来构建其他门电路和芯片，而不用去担心它们的内部结构。在我们开始构建计算机体系结构的起步阶段，仅仅只使用

原始门 Nand 来构建所有硬件。我们现在开始勾勒这个自底向上的硬件构建项目中的第一步，每次实现一个门。

我们故意只部分实现门电路，目的是让读者自己开发门电路的体系结构。这里再次重申：每个门电路都能够采用不止一种方式来实现；越简单的实现方法越好。

Not： 应用一个 2 位 Nand 门来实现一个 1 位 Not 门是很容易的。

小提示：请大家考虑正向思维。

And： 跟 Not 门一样，其实现也是很容易的。

小提示：请大家考虑逆向思维。

Or/Xor： 这些函数能够根据前面所实现的布尔函数来定义，使用一些简单的布尔操作。因此，这些门能够使用前面已经构建好的门电路来实现。

Multipleoxr/Demultiplexor： 同样，这些门能够使用前面已经构建好的门电路来实现。

多位 Not/And/Or 门（Multi-Bit Not/And/Or Gates）： 既然我们已经知道了如何来实现基本的 Not/And/Or 门，那么对 *n* 位门的实现就是将 *n* 个基本门组成阵列的过程，每个门独立地处理各自对应的位。这项工作可能有点繁琐，但是当这些多位门应用在复杂芯片上时，这项工作就显得很重要了，这一点将会在后面的章节具体介绍。

多位 Multiplexor（Multi-Bit Multiplexor）： *n* 位的 multiplexor 的实现是简单地将相同的选择位赋予 *n* 个二元 multiplexor。这个繁琐的工作同样可以构建非常有用的芯片。

多位门（Multi-Way Gates）： 实现小提示：想象一下吃饭的叉子。

1.4 观点 Perspective

本章介绍了应用数字设计项目中的第一步。在下一章会使用本章所实现的门电路来构建更复杂的功能模块。尽管我们选择 Nand 门作为基本构建模块，实际上其他的一些方法

也是可行的。例如，可以只使用 Nor 门来构建完整的计算机平台，或者综合使用 And 门、Or 门和 Not 门。这些逻辑设计的方法从理论上来说是相同的，就像几何学里面所有的理论都可以由一些持相反观点的公理来建立一样。关于理论和实际构建方法可以参看**数字设计**（*digital design*）和**逻辑设计**（*logic design*）相关的教材。

纵观本章，我们并没有去考虑设计的效率问题，比如在构建复合门电路时，设计中涉及到所使用的基本门的数量或者交叉线路的数量。这些问题在实际的设计生产中是极其重要的，大量的计算机专家和电子工程师们都在研究如何优化它们。另外一个根本没有提到的问题是门电路和芯片具体的物理实现，比如嵌入到硅上面的晶体管。当然有很多实现方式上的选择，每种方式都有它自己的特点（速度、供电、生产成本等等）。要了解这些问题需要电子学和物理学上的背景。

1.5 项目 Project

目标 实现本章提到的所有逻辑门。你能使用的构建模块只能是原始的 Nand 门以及由此所构建的一些复合门。

资源 这个项目里你所需要的唯一工具是与本书配套的硬件仿真器。所有的芯片应该利用 HDL 语言（参看附录 A）来描述。对于本章中提到的每种芯片，我们都提供了只有框架没有具体细节实现的`.hdl`骨干程序（文本文件）。另外，我们还提供了`.tst`脚本文件，用来告诉硬件仿真器如何进行测试，同时脚本文件将会产生一个正确的`.cmp`输出文件或称“比较文件（compare file）”。你要做的就是完成空缺`.hdl`程序里面的具体实现部分。

约定 当你设计的芯片（用`.hdl`程序表示）被加载到硬件仿真器中，并利用`.tst`文件进行测试后，应该在`.cmp`文件中生成输出列表。若结果不如所料，仿真器会给出相应的提示。

提示 Nand 门被认为是原始的逻辑结构，所以没有必要去构建它：只要你在 HDL 程序中使用 Nand 门，仿真器就会自动地调用其内置的 `tools/builtIn/Nand.hdl` 工具。我们建议你在这个项目中按照本章介绍的顺序来实现其他的门。因为 `builtIn` 目录下包括本书介绍的所有芯片模块，你可以直接使用而不用去定义它们：仿真器会自动调用其内置版本。

例如，来看一个项目里面提供的 `Mux.hdl` 骨干程序：假设因为某种原因，你没有完成程序编写，但是仍然希望在其他芯片设计中将 Mux 门作为内部模块使用。这不成问题，因为（应该感谢这个约定）：如果仿真器在当前路径下没有找到名为 `Mux.hdl` 的文件，它将自动调用内置的、由仿真器的软件提供的 Mux 模块。该内置实现（即存放在 `builtIn` 路径下的一个 Java 类）具有跟本书中描述的那些 Mux 门相同的接口和功能。因此，如果想让仿真器忽略一个或多个你自己的实现，只需要将相应的 `.hdl` 文件从当前路径下移除就行了。

建议按照如下步骤来进行：

0. 本项目需要用到的**硬件仿真器**放在与本书配套的软件包中（可从本书网站上下载）的 `tools` 目录里面。

1. 请阅读附录 A 的 A.1 至 A.6 小节。

2. 阅读硬件仿真器使用教程的第 I、II、III 部分。

3. 构建并仿真 `projects/01` 目录里面的所有芯片。

第2章 布尔运算

Boolean Arithmetic

Counting is the religion of this generation, its hope and salvation.
计算是这一代的信仰、希望和救赎。
——Gertrude Stein（1874 ~ 1946），美国作家

本章构建一种门逻辑设计的方案，该设计用来表达数字概念并能对数字进行算术操作。我们的出发点是在第一章中构建的一组逻辑门电路，本章中最终将会构建具有完整功能的算术逻辑单元。ALU 是核心的单元，它执行所有的算术和逻辑操作。因此，要理解中央处理器（Central Processing Unit, CPU）以及整个计算机的工作方式，构建 ALU 功能模块是很重要的一步。

跟前面一样，我们还是逐步来完成这个任务。第一部分简单介绍了相关**背景知识**，即如何利用二进制码和布尔运算来表达有符号数及其加法。**规范详述**部分介绍了一系列加法器芯片，它用来进行两位、三位，以及 *n* 位数字的加法。由此引出了 ALU 的规范详述部分，该规范基于一种成熟但简单的逻辑设计。**实现和项目**部分提供了如何利用与本书配套的硬件仿真器，在个人计算机上构建加法器芯片和 ALU 的思路和技巧。

二进制加法是简单而基本的操作，大部分数字计算机执行的操作能够简化为基本的二进制数的加法。因此，要实现大量依赖于二进制加法的计算机操作，其关键是要从构造的角度来理解二进制加法。

2.1 背景知识
Background

二进制数 十进制是以 10 为基底，而二进制系统则是以 2 为基底的。当我们碰到二进制数“10011”，并且被告知它代表整数时，这个数的十进制值是按以下方式来计算的：

$$(10011)_{two} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19 \qquad (1)$$

一般说来，令 $x = x_n x_{n-1} \ldots x_0$ 是一串数字。以 b 为基底的 x 值表示为 $(x)_b$，定义如下在基 b 中 x 的值，表示与，按如下方式定义：

$$(x_n x_{n-1} \ldots x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i \qquad (2)$$

读者可以用$(10011)_{two}$来验证上面的定义，我们看到式（2）的表示方法在 $b = 2$ 的情况下正好就是和式（1）等价。

式（1）的结果恰好是 19。因此，在运行一个电子制表程序时，分别敲击键盘上的 `1`、`9` 和 `Enter` 键，那么在计算机的存储器中的某一个寄存器里会储存对应的二进制数 10011。更精确地说，如果计算机正好是一个 32 位机，那么在寄存器里面存储的数就会是 00000000000000000000000000010011。

二进制加法 两个二进制数能够从右至左一位一位相加，跟十进制加法一样。首先，把两个数的最右边的一位（也称为 LSB，Least Significant Bits）加起来。然后，将两个位相加后的进位（0 或者 1）再与两个数的右起第二位相加。就按照这种方式计算下去，直到两个数最左边的一位（MSB，Most Significant Bits）为止。如果最后的两位数位相加后产生了进位 1，那么就说它产生了溢出；否则就说加法运算成功执行：

0	0	0	1		（进位）	1	1	1	1		
	1	**0**	**0**	**1**	x		**1**	**0**	**1**	**1**
+	**0**	**1**	**0**	**1**	y	+	**0**	**1**	**1**	**1**
0	1	1	1	0	$x + y$	1	0	0	1	0
	无溢出						有溢出			

可以看到用于两个 n-位数字二进制加法的计算机硬件可以由三位（两个计算位加上一个进位）加法的逻辑门构建而成。将进位转到下两个位上的加法是很容易实现的，只需通过 3 位加法器门电路的正确引线即可实现。

有符号二进制数 n 位二进制系统可以产生 2^n 个不同的组合。如果必须用二进制码表示有符号数，有个简单的方法就是，将这个空间分成两个相同的子集，一个子集用来表示正数，另一个表示负数。从理想的角度来说，编码规则应该按照如下原则来进行：有符号数的引入应该使硬件实现的复杂程度尽可能小。

有符号数的二进制表示促进了很多编码体系的发展。如今几乎所有计算机都采用称为**2-补码**（*2's complement*）的编码方式，也称为**基补码**（*radix complement*）。在 n-位的二进制系统中，数 x 的 2-补码定义如下：

$$\bar{x} = \begin{cases} 2^n - x & if \quad x \neq 0 \\ 0 & otherwise \end{cases}$$

举例来说，在一个 5-位二进制系统中，-2 的补码表示或称“$\text{minus}(00010)_{two}$”的补码表示为 $2^5 - (00010)_{two} = (32)_{ten} - (2)_{ten} = (30)_{ten} = (11110)_{two}$。为了验证该计算是否正确，读者可以验证 $(00010)_{two} + (11110)_{two} = (00000)_{two}$。注意，在后面的计算中，总和实际上是 $(100000)_{two}$，但是因为我们是在 5-位二进制系统中，所以最左边的第 6 位被忽略掉了。通常，当补码表示法应用在 n-位数字时，$x + (-x)$ 总是加到 2^n（即 1 后面跟 n 个 0）。图 2.1 显示了用补码表示的 4-位二进制系统。

通过观察图 2.1，可知用补码表示 n-位二进制系统有如下属性：

正	数	负	数
0	0000		
1	0001	1111	−1
2	0010	1110	−2
3	0011	1101	−3
4	0100	1100	−4
5	0101	1011	−5
6	0110	1010	−6
7	0111	1001	−7
		1000	−8

图 2.1　在 4-位二进制系统中的有符号数的 2-补码表示

- 系统能对所有 2^n 个有符号数进行编码，最大的数和最小的数分别为 $2^{n-1}-1$ 和 -2^{n-1}。
- 所有正整数的编码的首位是 0。
- 所有负整数的编码的首位是 1。
- 为了通过 x 的编码获得 $-x$ 的编码，所有最右边的 0 和从左起的第一个 1 保持不变，然后将剩余的位取反。等价的捷径就是，对 x 的所有的位取反，然后再加上 1，这个方案更容易在硬件中实现。

这种表示法有个特别有吸引力的特征：任何两个用补码表示的有符号数的加法和与正数的加法完全相同。比如，加法操作 $(-2)+(-3)$，使用补码（用一个 4-位表示），则要表示成 $(1110)_{two}+(1101)_{two}$。我们不用去关心这些代码表示的数字（正数还是负数），这个加法会得到结果 1011（丢掉了溢出位以后）。如图 2.1 所示，它正好是－5 的补码表示法。

简单来说，我们发现补码表示法可以实现任何两个有符号数的加法而不需要特殊的硬件。那么减法是怎样的呢？前面说过，在补码表示里面，对有符号数 x 的取反，也就是计算 $-x$，要将 x 的所有位取反然后再加 1。因此，减法可以被看成 $x-y=x+(-y)$。这样，硬件的复杂度也保持在最小。

这些理论结果的实际含义是很重要的。基本上，它们意味着能用单一芯片（称为**算术逻辑单元**，ALU 即 *Arithmetic Logical Unit*）将硬件执行的所有基本算术操作和逻辑操作都封装起来。现在来详细讨论这种 ALU，首先来介绍**加法器**（*adder*）电路。

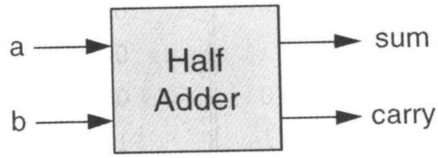
2.2 规范详述 Specification

2.2.1 加法器 Adders

我们介绍三个加法器，并由此引出多位加法器芯片：

- **半加器**（*Half-adder*）：用来进行两位加法。
- **全加器**（*Full-adder*）：用来进行三位加法。
- **加法器**（*Adder*）：用来进行两个 n-位加法。

输入		输出	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0




```
芯片名: HalfAdder
输  入: a, b
输  出: sum, carry
功  能: sum = LSB of a + b
        carry = MSB of a + b
```

图 2.2 半加器，用于 2-位二进制数的加法

我们也介绍了一种特殊的加法器，称为**增量器**（*incrementer*），用来对指定的数字加 1。

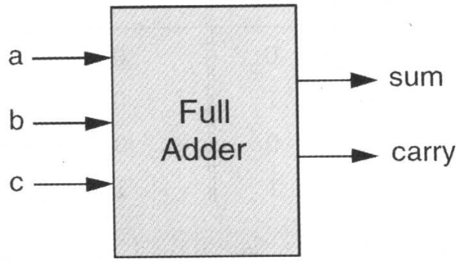
半加器 进行二进制数加法的第一步就是要能够对两个二进制位进行相加。我们把结果的 LSB 位称为 `sum`，MSB 位称为 `carry` 。图 2.2 展示了这种电路。

全加器 现在已经知道了如何对两个位进行相加，图 2.3 显示了**全加器电路**，用来对三个位相加。跟半加器一样，全加器电路也会产生两个输出：加法的 LSB 位和进位。

加法器 存储器和寄存器电路用 *n*-位的形式来表示整数，*n* 可以是 16、32、64 等等——这依赖于所在的计算机平台。进行 *n*-位加法的芯片称为多位加法器（multi-bit adder），或者简称为加法器。图 2.4 展示了一个 16-位加法器，对于任何 *n*-位加法器，相同的逻辑和表示按比例增加。

增量器 专门构建一个“对指定数字加 1”的电路，这样做会带来很多便利。这里给出了一个 16-位增量器的描述。

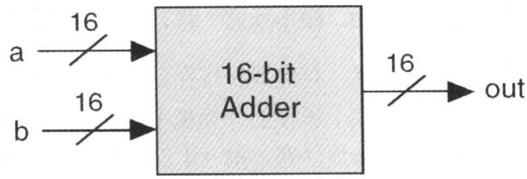
a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



芯片名：FullAdder
输　入：a, b, c
输　出：sum, carry
功　能：sum = LSB of a + b + c
　　　　　carry = MSB of a + b + c

图 2.3　全加器，用于 3-位二进制数的加法

...	1	0	1	1	a	+
...	0	0	1	0	b	
...	1	1	0	1	out	



芯片名：Add16
输　入：a[16], b[16]
输　出：out[16]
功　能：out = a + b
说　明：2 补码的整数加法。不处理溢出的情况。

图 2.4　16-位加法器，两个 *n*-位二进制数的加法

芯片名： Inc16
输　入： in[16]
输　出： out[16]
功　能： out=in+1
说　明： 2 补码的整数加法。不处理溢出的情况。

2.2.2 算术逻辑单元(ALU)

The Arithmetic Logic Unit (ALU)

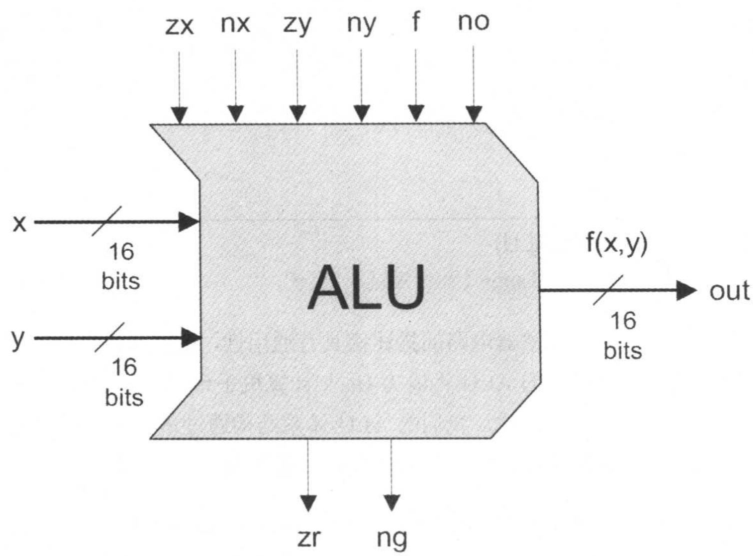
前面所介绍的对加法器电路的描述都具有通用性，也就是说它们适用于任何计算机。相比之下，这一小节描述的 ALU 将成为 Hack 计算机平台的核心部分。同时，ALU 的设计原则是相当通用的。不仅如此，我们的 ALU 体系结构通过最小的逻辑单元来实现大量功能，从这点来看，它是逻辑设计中高效和优良设计的典范。

Hack 的 ALU 计算一组固定的函数 $out = f_i(x,y)$，这里 x 和 y 是芯片的两个 16-位输入，*out* 是芯片的 16-位输出，f_i 是位于一个函数表中的函数，该函数表由 18 个固定函数组成。我们通过设置六个称为控制位（*control bits*）的输入位来告诉 ALU 用哪一个函数来进行何种函数计算。图 2.5 给出了用伪代码表示的详细的输入/输出规范。

要注意的是，这 6 个控制位的每一位指示 ALU 来执行某个基本操作。这些操作的各种组合可以让 ALU 计算多种有用的函数。因为全部操作都是由 6 个控制位引起的，那么 ALU 可以对 2^6=64 个不同的函数进行操作。图 2.6 列出了这些函数中的 18 种。

可以看到，我们是通过将 6 个控制位设置成函数 $f(x,y)$所对应的编码，来指示 ALU 计算该函数。从这点来看，图 2.5 描述的 ALU 内部逻辑应该产生如图 2.6 中列出的 $f(x,y)$的输出值。当然，这不是碰巧产生的，而是精心设计的结果！

比如，来看图 2.6 中的第 12 行，ALU 此时被指示要计算函数 `x-1`，那么 `zx` 和 `nx` 位是 0，这样 `x` 输入既不会变成 0，也不会被取反；`zy` 和 `ny` 位是 1，所以 `y` 输入首先按位变成 0，然后被按位取反。对 0 即 $(000\ldots00)_{two}$ 按位取反，得到 $(111\ldots11)_{two}$，正好是－1 的补码。




```
芯片名: ALU
输  入: x[16], y[16],     // 两个16位数据输入
        zx,               // x输入置零
        nx,               // x输入取反
        zy,               // y输入置零
        ny,               // y输入取反
        f,                // 功能码: 为1则代表Add, 为0则代表And
        no                // out输出取反
输  出: out[16],      // 16位输出
        zr,           // 若out=0则为True, 否则False
        ng            // 若out<0则为True, 否则False
功  能: if zx then x = 0           // 16位常量0
        if nx then x = !x          // 按位取反
        if zy then y = 0           // 16位常量0
        if ny then y = !y          // 按位取反
        if f then out = x + y      // 2补码的整数加法
        else out = x & y           // 按位与运算(And)
        if no then out = !out      // 按位取反
        if out=0 then zr = 1 else zr = 0   // 16位eq.比较
        if out<0 then ng = 1 else ng = 0   // 16位neg.比较
说  明: 不处理溢出的情况。
```

图2.5 算术逻辑单元(ALU)

这些位指示如何预设 x 输入		这些位指示如何预设 y 输入		此位用来选择是+还是And	此位指示如何设置out	生成的ALU输出
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

图 2.6 ALU 真值表。前六列二进制码表示的操作执行最右边列的函数（此处用符号!，&，和 | 来分别表示 Not，And 和 Or）。完整的 ALU 真值表包含 64 行，这里只列出了其中的 18 行

因此 ALU 的输入最后就是 x 和−1。因为 f 位是 1，选择的操作是算术加法，这使得 ALU 计算 x+(-1)。最后，no 位是 0，所以输出不会被取反。因此，最后 ALU 得到 x-1，正好达到我们的要求。

图 2.6 中描述的 ALU 逻辑对列出的其他 17 个函数的计算是否也是符合表中右边的函数表达式呢？为了验证它的正确性，读者可以试着从中选出几行来进行计算，验证其各自的 ALU 操作。注意到这些计算中的一些操作，比如 $f(x,y)=1$，它们也是有用的。还有一些其他有用的函数没有列在图中。

对这个特殊 ALU 的设计过程的描述是很有启发意义的。首先，我们制作了一张函数列表，其中包含了所有我们希望计算机能够执行的原始函数。接着，应用倒推法得出如何在二进制模式下操作 *x*, *y* 来实现期望的操作。这些处理要求以及“使 ALU 逻辑尽可能简单”的目标，导致我们的在设计中使用 6 个控制位，每一位都直接对应一个简单的二进制操作。这样的 ALU 构造简单且结构良好。从硬件的角度来说，简单的、良好的结构意味着便宜但功能强大的计算机系统。

2.3 实现
Implementation

我们的实现方针是部分实现，因为我们希望读者能自己去探索实际的芯片结构。通常，每个芯片都能通过多种途径实现；还是那句话，越简单的方法越好。

半加器 通过观察图 2.2 可以发现函数 sum(*a*,*b*)、carry(*a*,*b*) 恰好分别和标准的布尔函数 Xor(*a*,*b*)、And(*a*,*b*) 效果相同。因此，可以直接利用前面已经构建的门，于是半加器的实现就变得简单了。

全加器 全加器芯片可以通过两个半加器芯片和一个额外的简单门来实现。当然，不通过半加器而直接去实现的方法也是可行的。

加法器 用两个 *n*-位总线来分别表示两个以补码形式出现的有符号数，它们的加法可以按位执行，从右至左，执行 *n* 步。在第 0 步，两个最低位相加，产生的“进位”位直接参与下一位的加法。这样一直操作到第 $n-1$ 步，也就是两个最高位的相加。注意，到每一步都包含着三个位的相加。因此，*n*-位加法器可以通过这样的方法来实现：产生 *n* 个全加器芯片，并且依次把“进位”位传入下一个全加器。

增量器 *n*-位增量器可以通过 *n*-位加法器来实现。

ALU 可以看到，我们的 ALU 经过精心地设计，利用简单的布尔运算在逻辑上实现所有期望的 ALU 操作。因此，ALU 的**物理实现**被简化了，只须根据它们的伪代码规范来实现这些简单的布尔操作即可。你所要做的第一步可能是根据 `nx` 和 `zx` 控制位来建立一个逻辑电路，以操纵一个 16-位的输入（也就是，电路应该有条件地将 16-位输入置 0 和取反）。这个逻辑可以被用来操纵 `x` 和 `y` 输入，也可以操纵 `out` 输出。“按位与（bitwise And）”操作和加法的芯片已经在本章和前面的章节里面构建了。因此，剩下要做的就是构建“根据 `f` 控制位在它们两者之间进行选择”的逻辑。最后，你还需要构建一个逻辑，即将所有其他的芯片整合到 ALU 中（当我们说“构建逻辑”时，其意思是“编写 HDL 代码” ）。

2.4 观点

Perspective

本章介绍的多位加法器的结构是标准结构，虽然没有去考虑效率问题。实际上，我们介绍的加法器的实现是相当低效的，因为当进位从最低位逐位传递到最高位的过程中会产生很长时间的延迟。这个问题可以通过执行一种特定的逻辑电路来缓解，该电路实现了称之为进位预测（carry look-ahead）的技术。因为加法在任何硬件平台上都是最普遍的操作，所以任何对这种底层操作的改进都能使计算机整体性能有质的飞跃。

在任何计算机里，软硬件平台的整体功能都是由 ALU 和运行在其上的操作系统共同决定的。因此，当设计新的计算机系统时，ALU 应该实现多少种功能的操作，本质上是个性价比的问题。一般原则是，算术和逻辑操作的硬件实现成本通常较高，但是性能较好。我们在本书中采用的设计原则是，让 ALU 实现较少的基本功能，而尽可能用软件去实现其他扩展功能。比如，我们的 ALU 并不支持乘法、除法和浮点计算。但是我们会在操作系统层来实现这些操作（以及更多的一些数学函数），这些会在第 12 章中介绍。

关于布尔运算和 ALU 设计的详细介绍可以在很多计算机体系结构的教材中找到。

2.5 项目
Project

目标 实现本章介绍的所有芯片。你可以使用的模块包括在前面章节中介绍的芯片，以及你逐渐构建起来的芯片。

提示 当你的 HDL 程序调用前面已经构建的芯片时，建议使用这些芯片的内置（built-in）版本。这样做是为了确保正确性并加速硬件仿真器的操作。使用芯片内置版本的简单方法是：保证你的项目路径仅仅包含属于当前项目的`.hdl`文件。

本项目中的其他步骤跟前面一章的项目步骤一样，只是最后一步应该被替换成“对`projects/02`路径下的所有芯片进行构建和仿真”。

第 3 章　时序逻辑

Sequential Logic

It's a poor sort of memory that only works backward.

记忆如此悲怜，只能回溯过去。

——Lewis Carroll（1832 ~ 1898），英国作家

在前两章里面构建的所有的布尔芯片和算术芯片都是**组合芯片**（*combinational chips*）。组合芯片计算那些“输出结果仅依赖于其输入变量的排列组合”的函数。这些相关的简单芯片提供很多重要的处理功能（像 ALU 一样），但是它们却不能维持自身的状态。计算机不仅要能计算值，而且还需要存取数据，因而这些芯片必须配备**记忆单元**（*memory elements*）[1]来保存数据，这些记忆单元是由**时序芯片**（*sequential chips*）组成。

记忆单元的实现是复杂的过程，涉及了同步、时钟和反馈回路。其中的大部分能够被封装到称为**触发器**（*flip-flop*）的底层**时序门**（sequential gate）中。我们会将这些触发器作为基本的构建模块来使用，详细介绍和构建所有典型的现代计算机所采用的记忆设备（memory devices），涉及的内容从二进制单元（binary cells）到寄存器（registers）[2]，再到存储块（memory banks）和计数器（counters）。由此可构建出完整计算机所需的芯片组，并且我们将在第 5 章来介绍完整计算机的构建。

背景知识部分会对时钟和触发器作一个简单介绍，然后介绍由它们构建的所有记忆芯片（memory chips）。接下来的两个部分将会详细介绍芯片规范及其实现。跟前面章节一样，本章中提到的所有芯片都能用与本书配套的硬件仿真器来进行构建和测试，只要按照本章最后的**项目**部分中的指示进行操作即可。

[1] 关于对“memory”概念的理解，请参见本书第 81 页的脚注。——审校者

[2] 触发器（flip-flop）是具有记忆功能的最小记忆单元，它能够存储一个基本的比特位（bit），我们在这里称之为“寄存器（register）”。可以同时存储若干比特位（设 w 位）的寄存器我们称之为“w-bit 寄存器（w-bit register）”。可以利用一组 w-bit 寄存器进一步构成内存（memory）。需要读者注意的是，这里的所谓“寄存器（register）”与后面章节中讨论的 CPU 内部使用的寄存器虽然在英文术语都是“register”，但是从物理构成上和功能上是不一样的，希望读者仔细品味其中的差别，避免将这里的“寄存器”概念与大家熟悉的“寄存器”概念等同起来。——审校者

3.1 背景知识 Background

“记住一些东西”这个行为本质上是具有时间依赖性的：你**现在**记住的一些东西一定是**之前**的记忆。因此，为了构建能够“记忆”信息的芯片，我们首先必须开发一些标准的方法来表示时间的进程。

时钟（Clock） 在大多数计算机里，时间的流逝是用主时钟（master clock）来表示的，它提供连续的交变信号序列。其精确的硬件实现通常基于振荡器（oscillator），其在两个信号值 0-1，或称“低电平-高电平（*low-high*，*tick-tock*）”之间交替变化。两个相邻的上升沿之间的时间间隙称为时钟的**周期**（*cycle*），每个时钟周期模塑一个离散时间单元。通过使用硬件电路，这个信号同时被传送到计算机平台的每个时序芯片中。

触发器（Flip-Flops） 计算机里最基本的时序单元是称为**触发器**的设备，它有几个种变体。在本书里我们使用称为**数据触发器**（*Data Flip-Flop*，DFF 或称 D 触发器）的变体，其接口包含 1 比特位输入和 1 比特位输出。另外，DFF 有个**时钟**输入，根据主时钟信号连续地交变。数据和时钟的输入使得 DFF 能够实现基于时间的行为 $out(t)=in(t-1)$，这里 *in* 和 *out* 是门的输入和输出值，*t* 是当前时钟周期。换句话说，DFF 简单地将前一个时间周期的输入值作为当前周期的输出。

由此可见，这种基本行为是所有计算机硬件维持自身状态的基础，从二进制单元到寄存器以至任意大的随机存取记忆单元（RAM）皆是如此。

寄存器（Registers） **寄存器**是具有记忆功能的设备，它能够“储存”或称“记忆”某一时刻的值，实现经典的存储行为 $out(t)=out(t-1)$。从另一个方面来说，DFF 仅能够输出它前一时钟周期的输入，也就是 $out(t)=in(t-1)$。这就告诉我们，可以通过 DFF 来实现寄存器，只需将后面的输出反馈到它的输入就可以了，这样生成的设备如图 3.1 的中间部分所示。如此一来，在任何时刻 *t*，这个设备的输出都会重现它在时刻 $t-1$ 的值。

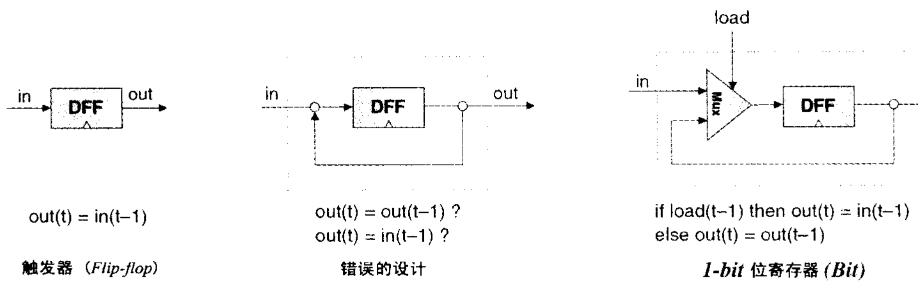


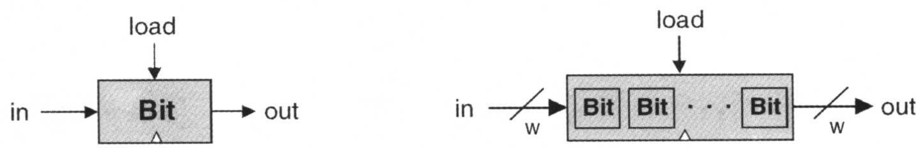
图 3.1 从 DFF 触发器到 1-bit 位寄存器。图中的小三角代表时钟信号的输入。用灰色小方块代表的芯片，以及用虚线框封装的整个芯片组都与时间存在相关性

然而，事实上并非如此。图 3.1 的中间部分所示的设备是不正确的。首先，因为无法告知 DFF 的输入什么时候从 `in` 获取，什么时候从 `out` 获取，所以我们不明确如何才能给这个设备输入新的数据值。一般来说，芯片设计的规则规定内部管脚的扇入（fan-in）必须为 1，也就是它们只能有一个单独的输入源。

这么规定的好处是，它让我们得到了正确的电路结构，如图 3.1 的右边部分所示，是解决我们输入不明确性问题的方法之一：在设计中使用**多路转换器**（*multiplexor*）。这个多路转换器的“选择位（select bit）”可以成为整个寄存器芯片的“加载位（load bit）”：如果希望寄存器开始存储一个新值，可以把这个值置于 `in` 输入口，然后将 `load` 位设为 1；如果希望寄存器一直储存它的内部值直到新的指令到来，可以将 `load` 位设为 0。

一旦实现了保存 1 比特位的基本机制，就可以轻松地构建任意位宽的寄存器，这可以通过由多个 1 比特位寄存器构成的阵列来实现，以构建可保存多比特位值的寄存器（如图 3.2 所示）。此类寄存器的基本设计参数是它的**宽度**，即它所保存比特位的数量——比如，16、32 或 64。此类寄存器的多位容量通常用**字**（*word*）来表示。

内存（Memories） 一旦具备了表示字的基本能力，就可以构建任意长度的存储块了。如图 3.3 所示，可以通过将很多寄存器堆叠起来形成随机存取 RAM 单元来实现。随机存取内存（RAM，*Random Access Memory*）这个概念由此得来：在 RAM 上能够随机访问被选择的字而不会受限于访问顺序。也就是说，我们要求内存中的任何字（无论具体物理位置在哪里）都能以相等的速度被直接访问。



if load(t–1) then out(t) = in(t–1)
else out(t) = out(t–1)

1-Bit 位的基本存储单元（*Binary cell*）

if load(t–1) then out(t) = in(t–1)
else out(t) = out(t–1)

w-bit 位寄存器（*w-bit register*）

图 3.2 从 1-bit 位寄存器到多 bit 位寄存器。存储宽度（width）为 *w* 的多 bit 位寄存器可以由 *w* 个 1-bit 位的寄存器组成。这两类寄存器的操作函数几乎一致，除了赋值操作（分别对 1-bit 位数值和多 bit 位数值进行赋值）“=”之外

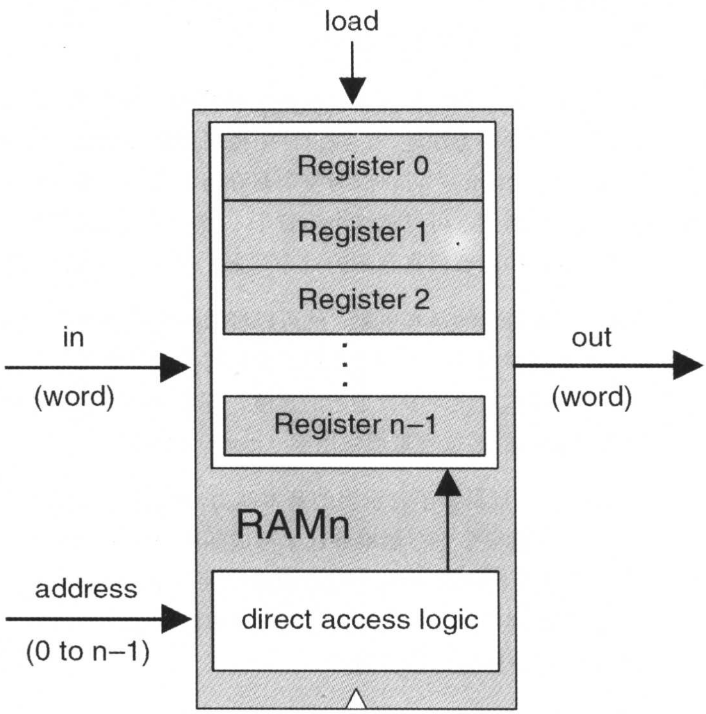


图 3.3 （概念上的）内存（RAM）芯片。RAM 的存储宽度和容量不定

这个规定可以通过以下方式实现。首先，根据其各自被存取的位置——也就是 n 位 RAM 中每个记忆单元——分配一个唯一的**地址**（*address*，一个从 0 到 $n-1$ 之间的整数）。然后，构建一个由 n-寄存器构成的阵列，再构建一个逻辑门，使得该逻辑门能够对给定的地址 j，找到地址 j 对应的记忆单元。要注意，由于记忆单元没有以物理实现的方式被标上地址，因此地址概念并没有在 RAM 设计中显式地表现出来，而我们在后面将会看到，当芯片配备了直接访问逻辑部件后，就能以逻辑方式实现寻址的概念。

总的来说，典型的 RAM 设备接收三种输入：数据输入、地址输入和加载位。**地址**指定了在当前时种周期里哪一个 RAM 寄存器被访问。进行读操作时（`Load = 0`），RAM 的输出立即发出被选中的记忆单元的值。在进行写操作（`load = 1`）时，被选择的记忆单元将在下一个时间周期内被赋予新输入值，从此 RAM 将开始发出该新输入值。

RAM 设备的基本设计参数是它的数据宽度（即每个字的宽度）和它的大小（RAM 中的字的数目）。现代计算机一般采用 32-位宽或者 64-位宽的 RAM。

计数器（Counter） 计数器是一种时序芯片，它的状态是整数，每经过一个时间周期，该整数就增加 1 个单位，执行函数 $out(t)=out(t-1)+c$，这里 c 就是 1。计数器在数字系统中担当非常重要的角色。比如，典型的 CPU 包括一个**程序计数器**（*program counter*），它的输出就是当前程序中下一步将要执行的指令地址。

计数器芯片可以通过将标准寄存器的输入/输出逻辑和附加的常数组合逻辑相结合来实现。一般来说，计数器必须要配一些附加的功能块，比如将计数置零、加载新的计数值，或者用减操作来取代增操作。

时间问题 到目前本章所介绍的芯片都是**时序芯片**（*sequential chip*）。简单地说，时序芯片就是直接或间接地嵌入一个或多个 DFF 门的芯片。从功能的角度来说，时序芯片被 DFF 门赋予了维持状态（如内存单元）或者对状态进行操作（如计数器）的能力。

从技术的角度来说，这是通过在时序芯片内部建立反馈回路（如图 3.4 所示）来实现的。在组合逻辑芯片中根本没有提到时间，在这里介绍反馈回路就有点麻烦：输出依赖于输入，而输入本身又依赖于输出，因此输出只依赖自身。另一方面，将时序芯片的输出反馈到输入是非常容易的，因为 DFF 中存在内在的时间延迟：在时刻 t 的输出不依赖于其当前输出值，而依赖于 $t-1$ 时刻的输出值。这一属性能避免带反馈电路的组合逻辑芯片中出现的"数字竞争[3]（data race）"。

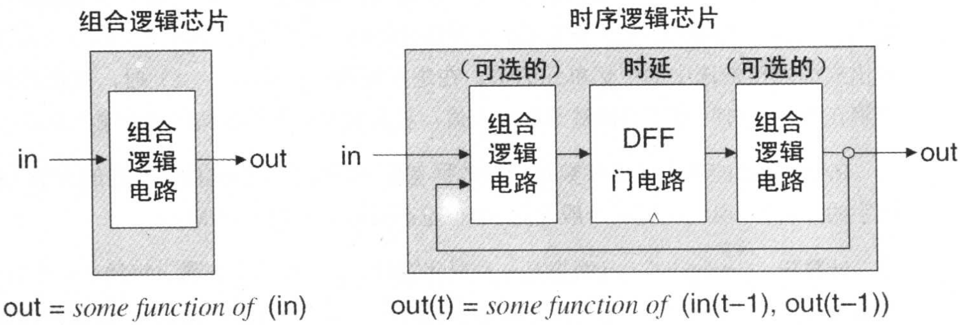


图 3.4　组合逻辑芯片与时序逻辑芯片的比较（in 和 out 可代表一个或多个输入/输出变量）。时序逻辑芯片总是像三明治一样（DFF 门电路夹在两个可选择的组合逻辑电路中间）

前面说到组合逻辑芯片的输出随其输入的变化而变化，而不去考虑时间。相比之下，时序结构中包含的 DFF 保证了它们的输出变化仅仅发生在一个时钟周期到下一个时钟周期的转换点上，而不在时钟周期之内。实际上，我们允许时序芯片在时钟周期**之内**出现不稳定的状态，但是必须保证在下一个时钟周期的起始点，其输出值是正确的。

时序芯片输出这种"离散化（discretization）"过程有个重要作用：它能被用来对整个计算机系统进行同步。例如，假设我们指示算术逻辑单元（ALU）计算 $x+y$，其中 x 是与 ALU 邻近的 RAM 寄存器的值，y 是距 ALU 较远 RAM 寄存器的值。由于各种物理条件上的限制（距离、阻力、干涉、随机噪声，等等），代表 x 和 y 的电信号可能会在不同的时刻到达 ALU。然而，对于**组合逻辑芯片**（*combinational chip*）而言，ALU 并没有时间概念——ALU 只负责不间断地把出现在输入端的值加起来。因此，ALU 的输出稳定到正确的 $x+y$ 结果需要一些时间。在结果值稳定之前，ALU 会产生垃圾值。

[3] "数字竞争（data race）"是数字电路技术中使用的术语。"数字"更突出了"硬件"上的概念，而"数据"更强调的是"逻辑"上的概念。一般书籍将"data race"译为"数字竞争"，故在此沿用这种译法。——审校者

如何来解决这个问题呢？ALU 的输出总是被发送到某种类型的时序芯片（寄存器、RAM 存储单元等等），具体什么芯片我们并不关心。只要保证：在设计计算机时钟时，时钟周期的长度要比 1 个比特在计算机系统中两个物理距离最长的芯片之间的传输时间稍长。这样就能在时间上保证时序芯片能够更新其状态（在下个时钟周期的开始处），也就是保证它从 ALU 那里接收到的输入值是有效的。简单地说，就是将一组相互独立的硬件组件同步为一个协调统一的系统，关于此还会在第 5 章中介绍。

3.2 规范详述 Specification

这一节介绍一组时序芯片：

- D 触发器（DFFs）
- 寄存器（基于 DFFs）
- 存储块（基于寄存器）
- 计数器芯片（也是基于寄存器）

3.2.1 D 触发器 Data-Flip-Flop

我们介绍的最基本的时序设备是 **DFF** 门（*data flip-flop gate*），也是用于设计所有记忆单元的基本组件。DFF 门包含 1 个比特位数据输入和 1 个比特位数据输出，如下所示：



芯片名： DFF
输　入： in
输　出： out
功　能： out(t) = in(t-1)
说　明： 该 DFF 门有现成的内置实现版本（built-in implementation），因此没有必要自己去实现。

跟 Nand 门一样，DFF 门在我们的计算机体系中处于非常底层。可以说，计算机中的所有时序芯片（寄存器、内存、计数器）都基于大量的 DFF 门。

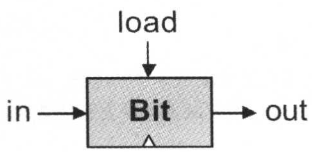
所有这些 DFF 门都连接同一个主时钟，形成巨大的分布式“合唱阵容（计算机系统中的所有时序芯片都在主时钟频率这个‘总指挥’的协调统一之下共同工作）”。在每个时钟周期的启始点，计算机中**所有** DFF 的输出都被赋予它们上个时钟周期的输入。在其他时间，DFF 被“锁存（latched）”了，这意味着它们的输入将暂时不会影响它们的输出。这个传导性操作将影响到组成系统的上百万个 DFF 门，大约每秒十亿次（依据计算机的时钟频率而定）。

通过同时向系统中所有 DFF 提供统一的主时钟信号，来实现计算机硬件对时间的相关性（time dependency），硬件仿真器能够以软件的方式模拟出相同的效果。只要计算机体系结构是确定的，那么仿真的结果就将是一致的：只要所设计的芯片中包含 DFF 门，那么整个芯片乃至在该硬件层之上构建的其他所有芯片都将继承对时间的相关性。按照定义，这些芯片都称为**时序芯片**（sequential chip）。

DFF 的物理实现是个复杂的任务，它使用反馈回路（feedback loops，仅基于 Nand 门的经典设计）来连接几个基本的逻辑门。本书将这个复杂过程忽略，并将 DFF 当作原始构建模块。因此，硬件仿真器提供了内置 DFF 实现供其他芯片使用。

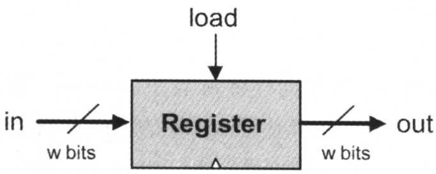
3.2.2 寄存器
Register

1 比特位寄存器，可以称为**比特**（*Bit*），或**二进制单元**（*binary cell*），用来存储信息的 1 个比特位（0 或 1）。芯片接口包括：一个输入管脚（它传输一个比特数据）；一个能够使该寄存器进行写操作的 `Load` 管脚；一个输出管脚（用来输出寄存器的当前状态）。二进制单元的接口图和 API 如下所示：




```
芯片名: Bit
输  入: in, load
输  出: out
功  能: If load(t-1) then out(t)=in(t-1)
        else out(t)=out(t-1)
```

寄存器芯片的 API 本质上时是一样的，除非输入管脚和输出管脚被设计成处理多位数据值：




```
芯片名: Register
输  入: in[16], load
输  出: out[16]
功  能: If load(t-1) then out(t)=in(t-1)
        else out(t)=out(t-1)
说  明: "=" 是16-位的赋值操作。
```

比特和寄存器芯片的读/写行为几乎是一样的：

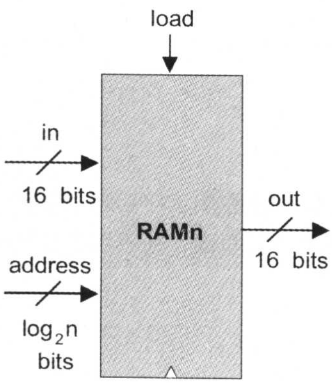
读（*Read*）：要读取寄存器的内容，只需获取它的输出。

写（*Write*）：要将新数据值 *d* 写入寄存器，只需将 *d* 置于 `in` 输入管脚，并将 `load` 输入管脚的值设为 1。这样，在下一个时钟周期内，寄存器被赋予了新的数据值，它的输出变成了 *d* 。

3.2.3 存储

Memory

可进行直接访问的记忆单元也称为 RAM，是由 *n* 个 *w*-位寄存器组成的阵列，配有直接访问电路（direct access circuitry）。寄存器的个数（*n*）和每个寄存器的宽度（*w*）分别称为内存的**尺寸**（*size*）和**宽度**（*width*）。我们将开始构建一组这样的存储体系，都是 16 位宽，但是大小不同：RAM8、RAM64、RAM512、RAM4K 和 RAM16K。所有这些内存芯片都有相同的 API，因此我们在参数化的图表里面描述它们。




```
芯片名: RAMn        //n和k在以下有说明
输  入: in[16], address[k], load
输  出: out[16]
功  能: out(t)=RAM[address(t)](t)
说  明: "=" 是16-位的赋值操作。
```

用于 Hack 平台的具体 RAM 芯片如下：

芯片名	*n*	*k*
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

读：要读取编号为 *m* 的寄存器，我们先将 *m* 置于 `address` 输入管脚。RAM 的直接存取逻辑将选中编号为 *m* 的寄存器，该寄存器于是将它的输出发送到 RAM 的输出管脚。这是个不涉及时间的组合逻辑操作。

写：要将新的数据值 *d* 写入编号为 *m* 的寄存器，我们同样将 *m* 置于 `address` 输入管脚，并将 *d* 置于 *in* 输入管脚，然后确认 `load` 位为 1。这样使得 RAM 的直接存取逻辑去选中 *m* 号寄存器，`load` 位来使能写操作。在下一个时钟周期里，被选中的寄存器将会被赋予新的数据值 *d*，RAM 的输出值也变成了 *d*。

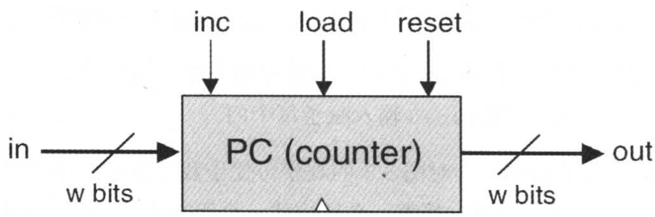
3.2.4 计数器
Counter

虽然**计数器**（*counter*）是独立的抽象，但是可以通过它的一些应用来描述。比如，考虑一个指令地址计数器，它的内容是计算机下一时钟周期要读取和执行的指令地址。在大多数情况下，计数器在每个时钟周期内做简单的“加 1”操作，因此计算机能够获取程序的下一条指令。在其他情况下，比如要“跳跃去执行编号为 *n* 的指令”时，我们希望能够将计数器设为 *n*，然后它继续进行默认的计数行为，$n+1$、$n+2$，等等。甚至，将计数器清零。在任何时候可通过将计数器置 0 来让程序重新执行。简单地说，我们需要一个可重载的、可重置的计数器。

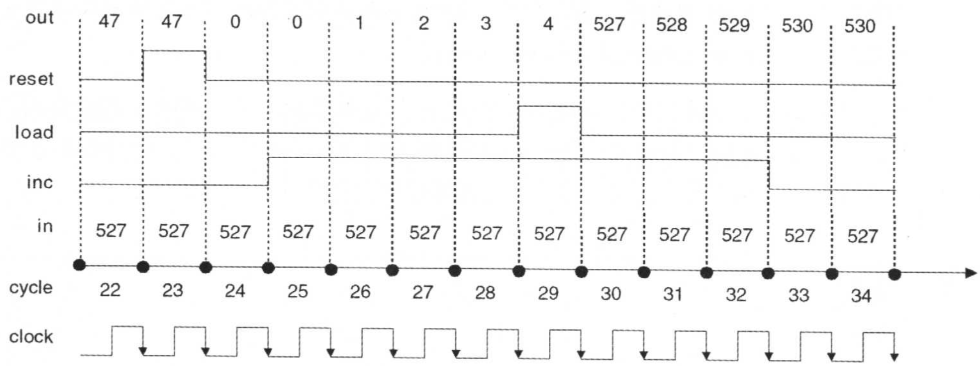
如此一来，我们的计数器芯片接口就与寄存器的接口十分相似，只是计数器芯片有两个附加的控制位 `reset` 和 `inc`。当 `inc=1` 时，计数器在每个时钟周期自加，输出值 `out(t)=out(t-1)+1`。如果想要将计数器重置为 0，就将 `reset` 位置为 1；如果想要将其初始化为某个计数值 *d*，就将 *d* 置于 `in` 输入管脚然后将 `load` 位置 1。具体细节在计数器 API 里面有描述，它的操作实例如图 3.5 所示。

3.3 实现
Implementation

触发器 DFF 门能够利用象第 1 章中介绍的一些底层级逻辑门来实现。然而，本书将 DFF 看作是原始门，因此它们可以直接被用在硬件构造项目里面，而不用去管这些 DFF 的内部结构。




```
芯片名: PC  //16-bit 的计数器
输  入: in[16], inc, load, reset
输  出: out[16]
功  能: If reset(t-1) then out(t)=0
        else if load(t-1) then out(t)=in(t-1)
             else if inc(t-1) then out(t)=out(t-1)+1
                  else out(t)=out(t-1)
说  明: “=”和“+”分别对 16-bit 位的数值进行赋值操作和加法操作。
```



我们假设从第 22 个时间周期开始观察计数器，这时它的输入和输出恰好分别是 527 和 47。还假设计数器的控制位（reset，load，inc）起始数值为 0。

图 3.5 计数器的模拟。在第 23 个时钟周期时出现一个 *reset* 信号，该信号使计数器在接下来的两个时钟周期内的输出值都为 0，因为第 25 个时钟周期出现了一个 *inc* 信号，这又使得计数器从下一个时钟周期开始作累加计数（incrementing）。计数工作持续到第 29 个时钟周期为止， 因为此时 *load* 位被置为 1。如图，计数器的输入端始终是 527，所以计数器从下个时钟周期开始就从 527 开始计数。整个计数过程中 inc 位必须始终为 1，所以计数器的计数工作将会在第 33 个时钟周期 inc 被置 0 时结束

1-位寄存器（Bit） 这个芯片的实现已经在图 3.1 中介绍过。

寄存器 由 1-位寄存器来构建 *w*-位寄存器是非常简单的。所须要做的就是构建一组 *w* 比特门，然后将寄存器的 `load` 输入赋予每个门。

8-寄存器（RAM8） 对图 3.3 的观察可便于我们理解。要实现 RAM8 芯片，就要将一组 8 个寄存器排列起来。接着，必须构建一个组合逻辑，把从 RAM8 “`in`” 输入端的输入值装载到被选中的某个记忆单元内。依此类推，还需构建一个组合逻辑，其对给定的 `address` 值选择正确的寄存器，然后将它的 `out` 输出值送到 RAM8 的 `out` 输出上。提示：这个组合逻辑已经在第 1 章中实现了。

***n*-寄存器** 任意长度（以 2 为幂）的存储块可以由一些小的记忆单元递归构建到单寄存器级而成，这个观点体现在图 3.6 中。集中考虑图的右边部分，可以发现 64-位寄存器 RAM 可以由 8 个 8-位 RAM 芯片构成。要从 RAM64 内存中选择某个特定的寄存器，我们使用形为 *xxxyyy* 的 6-位地址。作为 MSB 的 xxx 位选择一个 RAM8 芯片，作为 LSB 的 *yyy* 位在选定的 RAM8 中选择一个寄存器。RAM 芯片内部必须配备必要的逻辑电路来实现这种阶层式寻址机制（hierarchical addressing）。

计数器 *w*-位计数器包含两个主要部分：一个常规的 *w*-位寄存器和组合逻辑。组合逻辑用来：(a) 执行计数功能；(b) 根据控制位的 3 种不同的命令值，将计数器置于正确的操作模式。提示：这个逻辑的大部分功能都已经在第 2 章中实现了。

3.4 观点 Perspective

本章介绍的所有内存系统的基础是触发器，并将这种门电路看作是原始的构建模块。在硬件知识的教材中，通常是从基本的组合逻辑门（比如 Nand 门）开始讲述，使用合适的反馈环来实现触发器。 对于标准的实现方法，首先要构建一个简单的（无时钟的）的

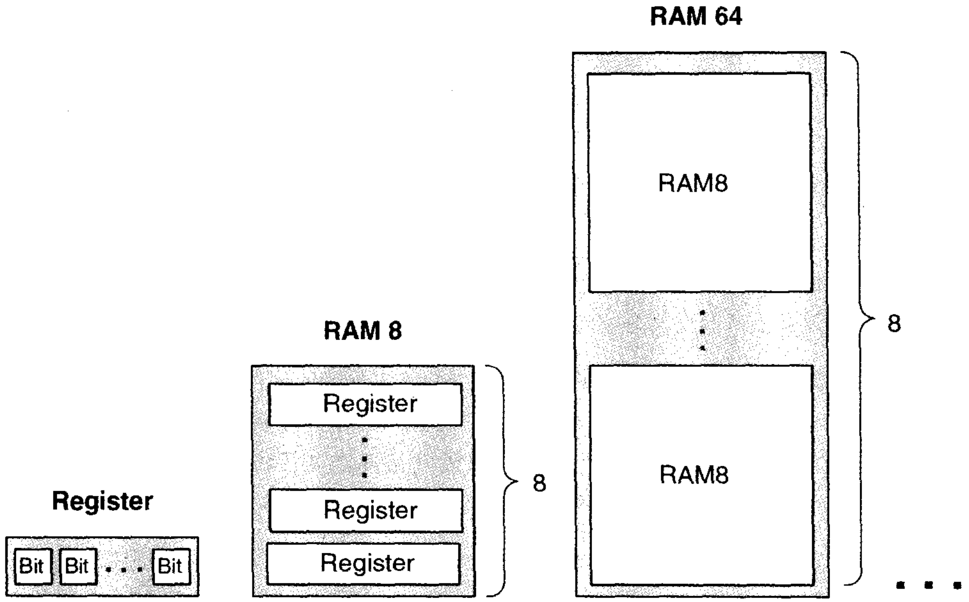


图 3.6 通过递归组合（recursive ascent）逐步构建内存块（memory blank）。一个 *w*-bit 寄存器由 *w* 个 1-bit 寄存器构成，RAM8 内存块由 8 个 *w*-bit 寄存器构成，RAM64 由 8 个 RAM8 芯片构成，依次类推。一个 16K RAM 芯片按这种方法继续往后三步就可构建完成

双态（bi-stable）触发器，也就是能被置于两种状态中的某一种。于是，带时钟的触发器则是通过两个这种简单的触发器的级联实现的，第一个触发器在时钟 ticks 时被设定，第二个在时钟 tocks 被设定。这种“主-从”设计使得整个触发器具有所期望的时钟同步功能。

这些构建工作是相当复杂的，需要对一些细节问题（比如组合逻辑电路里面的反馈环的影响，以及利用双态的二进制时钟信号来实现时钟周期）有较好的理解。本书将触发器看作是一个原始门，这样就不用去考虑其底层实现了。有兴趣了解触发器电路内部结构的读者可以参看大多数逻辑设计和计算机体系相关的教材。

最后，应该说现代计算机的存储器设备并不总是由标准的触发器构建而成。利用内在存储技术的独特物理属性，现代的存储器芯片通常都被精心地优化。对于计算机设计人员而言，有很多可供选择的技术；采用何种技术仍然是个性价比的问题。

除了这些针对底层的考虑以外，本章讲述的所有其他芯片——基于触发器之上而构建的寄存器和内存芯片——都是标准的芯片。

3.5 项目 Project

目标　构建本章所有的芯片。你可以使用的模块是原始的 DFF 门及构建于其上的芯片，以及前面章节里面介绍的芯片。

资源　本项目所需要的唯一工具就是与本书配套的硬件仿真器。所有的芯片应该用附录 A 中的 HDL 语言来实现。跟前面一样，对于每个芯片，我们提供一个骨干`.hdl`程序（即没有给出具体实现部分），另外还有供硬件仿真器进行仿真测试的`.tst`脚本文件，以及`.cmp`比较文件。你的工作就是要完成`.hdl`程序的实现部分。

约定　当程序被加载到硬件仿真器中后，你的芯片设计（`.hdl`程序）应由`.tst`文件来进行测试，并在`.cmp`文件中产生输出。若结果不如所料，仿真器会给出相应的提示。

提示　D 触发器（DFF）门被认为是原始门，因此没有必要去构建它：当仿真器在 HDL 程序中遇到 DFF 门时，它会自动调用内置在`tools/builtIn/DFF.hdl`中已经实现的 DFF 门模块。

本项目的路径结构　用较小的 RAM 芯片构建较大的 RAM 芯片时，推荐使用前者的内置（built-in）版本。否则，仿真器可能会运行得非常缓慢，甚至溢出内存空间，因为大 RAM 芯片包含成千上万的底层芯片，在仿真过程中，所有这些芯片的实现都以软件对象的形式被仿真器装入计算机内存。基于这个原因，我们将`RAM512.hdl`、`RAM4K.hdl`、`RAM16K.hdl`程序放在单独的路径里。这样，对 RAM4K 和 RAM16K 芯片的向下递归构建是到 RAM512 芯片为止的，因为构建 RAM512 芯片的更底层芯片是内置的（仿真器在这个路径下找不到它们的程序）。

步骤 推荐按照下面的顺序来进行：

0. 本项目需要的硬件仿真器在软件包的 `tools` 路径下可以找到。
1. 阅读附录 A，尤其的是 A.6 和 A.7。
2. 浏览硬件仿真器指南，尤其是第 IV 和 V 部分。
3. 构建和仿真在 `project/03` 目录下给出的所有芯片。

第 4 章　机器语言

Machine Language

Make everything as simple as possible, but not simpler.

让每件事情尽可能简单，而又不简单过度。

——阿尔伯特·爱因斯坦（1879 ~ 1955）

我们可用**结构化**（constructively）的方式来描述计算机，给出它的硬件平台，然后解释它是如何由一些底层芯片构建而成的。也可以通过描述和演示机器语言的功能来抽象地描述计算机。事实上，要开始了解一个新计算机系统，先去阅读一些用其机器语言编写的底层程序是十分必要的。这样不仅有助于了解如何通过程序让计算机去执行正确的操作，还可以由此了解为什么其硬件会按某种特定的方式来设计。基于这种思想，本章集中讨论用底层机器语言进行编程，从而为第 5 章打下基础。在第 5 章里，我们会构建一个完整的，能运行机器语言的通用计算机。该计算机将由第 1 至 3 章中介绍的芯片构建而成。

机器语言是一种约定的形式，用来对底层程序进行编码，从而形成一系列机器指令。应用这些指令，程序员可以命令处理器执行算术和逻辑操作，在内存中进行存取操作，让数据在寄存器之间传递，验证布尔表达式的值，等等。高级语言的基本设计目标是通用性和较强的表达力；与高级语言相反，机器语言的设计目标是能直接运行在指定的硬件平台上，并能对这个平台进行全面操控。当然，通用性、表达力和良好的结构仍然是必要的，但这些是仅相对于它所支持并直接运行于其上的硬件平台而言的。

机器语言是整个计算机体系中意义最深奥的接口——它也是硬件和软件相接的中间线。借由机器语言，程序员用符号指令表达的抽象思维被转换成执行在硅片上的物理操作。

因此，既可以将机器语言看作编程工具，也可以将其看作硬件平台内部不可分割的一部分。事实上，正如我们设计机器语言是为了使用给定的硬件平台一样，我们设计硬件平台是为了获取、解析并执行用给定机器语言编写而成的指令。

本章首先对机器语言编程做简要的介绍，然后对 Hack 机器语言进行详细描述，包括其二进制版本和符号汇编（symbolic assembly）版本。本章末尾的练习项目就是让你来编写一些机器语言程序。该项目能让你对底层（low-level）的编程有个比较好的理解，为你下一章构建计算机系统做准备。虽然大多数人以后永远不会直接用机器语言来编写程序，但是学习底层编程是彻底理解计算机体系的必备条件。而且，你的发现是多么激动人心：最复杂的软件系统，在底层其实是一长串基本指令，每一条指令都描述了底层硬件上的一种极为简单而原始的操作。还是那句老话，要达到这样的理解程度，最好是动手写一些底层代码并让其直接在硬件平台上运行。

4.1 背景知识 Background

本章是面向语言来进行阐述的，因此我们在这里可以将大多数底层硬件平台的细节进行抽象，予以忽略，将具体的细节放在下一章来讨论。实际上，为了对机器语言做一般性描述，我们只须要集中讨论三个主要的抽象体上：**处理器**（processor）、**内存**（*memory*），以及一组**寄存器**（*registers*）。

4.1.1 机器 Machines

机器语言（*machine language*）可以被看作是一种约定的形式，它利用**处理器**和**寄存器**来操控**内存**。

内存　内存（memory）的概念是指“用来储存数据和指令的硬件设备”。从程序员的观点来看，所有的内存具有相同的结构：一个连续的固定宽度的单元序列，也称为**字**（word）或**内存单元**，每个内存单元都一个唯一的**地址**（*address*）。因此，对于独立的字（word，代表一个数据项或是一个指令），可以通过提供它的地址来描述。

简便起见，在后面的内容中我们将会使用符号 Memory[address]，RAM[address]，或者 M[address] 来表示内存。

处理器 处理器，通常又称为**中央处理器**或 CPU（*Central Processing Unit*），是执行一组固定基本操作的设备。这些操作通常包括算术操作和逻辑操作，内存存取操作和控制操作（*control operation*，也称 *branching operations*）。这些操作的对象是二进制数值，它们来自寄存器和指定的内存单元。类似的，操作的结果（处理器的输出）既可以存储在寄存器内，也可以存储在指定的内存单元。

寄存器 内存访问是相对较慢的操作，需要很长的指令格式（一个地址可能需要 32 位）。基于此原因，大多数处理器都配有一些寄存器，每个寄存器只存储 1 位。它紧挨着处理器，相当于处理器的一个高速本地内存，使得处理器能快速地操控数据和指令。寄存器使得程序员能够尽可能地使用内存访问命令，从而加速程序的执行。

4.1.2 语言
Languages

机器语言程序是一系列的编码指令。比如说，在 16-位计算机上的典型指令之一是 `1010001100011001`。为了知道这个指令的意思，就必须知道语言的规则，也就是底层硬件平台的指令集。例如，这样的指令包含四个 4 比特的位域（fields）：最左边的域是 CPU 的操作编码，剩下的三个部分表示该操作的操作数。因此根据该硬件平台上的机器语言语法，前面的命令代表 *set R3 to R1+R9*。

鉴于二进制码相当晦涩，通常会在机器语言中同时使用二进制码和**助记符**（*symbolic mnemonics*）。助记符是一种符号标记，它的名字暗示了其所代表的意思——硬件元素和二进制操作。例如，语言设计者可定义操作码 `1010` 用 `add` 来表示，机器中的寄存器可以使用符号 `R0`、`R1`、`R2` 等来表示。我们可以直接地描述机器语言指令，比如 `1010001100011001`，用助记符表示为 `Add R3,R1,R9`。

将这种符号抽象更进一步发展，我们不仅能够**阅读**符号表示，而且能实际地利用这些符号命令而不是二进制指令来**编写**程序。接下来，可以使用文本处理程序，将这些符号命令解析为其内含的意域（助记符或操作数），将每个意域翻译成其对应的二进制表示，然后将生成的代码汇编成二进制机器指令。符号表示也称为**汇编语言**（*assembly language*），或简单地说成汇编，而将汇编程序翻译成二进制码的程序则称为**汇编编译器**（*assembler*）。

因为不同的计算机在 CPU 的操作方式、寄存器的数量和类型，以及汇编语法上各不相同，活像是机器语言的巴别塔传说[1]，每种计算机都有它自己的晦涩语法。然而抛开差异性，所有的机器语言都支持相似的通用命令集合，接下来会介绍。

4.1.3 命令
Commands

算术操作和逻辑操作 计算机需要执行基本的算术操作（比如加法和减法），以及基本的布尔操作（比如按位取反、移位等等）。这里给出了一些例子，它们是使用典型的机器语言语法编写的：

```
ADD R2,R1,R3    // R2 ← R1+R3 其中 R1,R2,R3 是寄存器
ADD R2,R1,foo   // R2 ← R1+foo 其中 foo 代表的意思是
                // 用户定义的标签 foo 所指向的
                // 内存单元的值
AND R1,R1,R2    // R1 ← 对 R1 和 R2 进行按位与（And）操作的结果
```

内存访问 内存访问命令分为两类。第一类，正如刚才看到的，算术命令和逻辑命令不仅允许操控寄存器，而且还可以操控特定的内存单元。第二类，所有的计算机都会使用 `load` 和 `store` 命令，用来在寄存器和内存之间传递数据。这些访问命令可能会应用某些类型的寻址方式，并在指令中指定目标内存单元的地址。当然，不同的计算机有不同的寻址方式，下面列出三种绝大多数计算机都支持的寻址方式：

- **直接寻址**（*Direct Addressing*） 最常用的寻址方式，直接表示一个指定内存单元的地址，或者使用一个符号来代表这个指定的地址，如下所示：

[1] 巴别塔，Tower of Babel，古语"上帝之门（gate of God）"的意思。旧约圣经（创世纪 11:1-9）中记载，挪亚（Noah）的后代们打算在示拿（Shinar，地名）的一片平原上建造一座通天塔，耶和华为了阻止他们建造巴别塔，就变乱他们的口音，使他们语言不通，由此使众人分散在了世界各地。——技术编辑

```
LOAD R1,67 // R1 ← Memory[67]
// 或者假设 bar 指向内存地址 67，那么就有：
LOAD R1,bar// R1 ← Memory[67]
```

- **立即寻址**（*Immediate Addressing*） 这种寻址方式被用来加载常数——也就是说，加载那些出现在指令代码里面的数值：我们直接将指令数据域中的内容当作要操作的数据装入寄存器，而不是将该数值当作内存单元的地址，如下所示：

```
LOADI R1,67 // R1 ← 67
```

- **间接寻址**（*Indirect Addressing*） 在这种寻址模式中，要访问的内存单元的地址没有直接出现在指令中，而是指令指定的内存单元中的内容代表目标内容单元的地址。这种寻址模式被用来处理**指针**（*pointer*）这种语言设施。比如，一条高级语言命令 x=foo[j]，这里 foo 是数组变量，x 和 j 是整数变量。那么与这条命令等价的机器语言是什么呢？当数组 foo 在高级语言程序里被声明并被初始化时，编译器分配一组连续的内存单元来保存这个数组数据，并用符号 foo 来指代该内存单元组的**基地址**（*base address*）。

于是当编译器后来遇到表示数组单元的符号（比如 foo[j]）时，将按以下步骤进行地址解析。首先，应注意第 *j* 个数组入口是某个内存单元的物理地址，该地址相对于数组基地址的偏移量为 *j*(为简单起见，假设每个数组元素占用一个字)。因此，表达式 foo[j] 相关的地址可以很容易地被计算出来，即只需将 foo 的值加上 *j* 即可。如在 C 程序语言中，x=foo[j]这样的命令也可以等价地表示成 x=*(foo+j)，这里“*n”代表“Memory[n]的值”。被翻译成机器语言时，这样的命令根据特定汇编语言的语法，会产生如下的代码：

```
// 将 x=foo[j] or x=*(foo+j) 翻译成汇编语言：
ADD R1,foo,j // R1 ← foo+j
LOAD* R2,R1  // R2 ← Memory[R1]
STR R2,x     // x  ← R2
```

控制流程 程序通常以线性方式，一个命令接着一个命令执行，但偶尔也包含分支，执行其他地方的命令。分支能够实现好几种结构，包括**反复**（*repetition*，跳回到循环的初始位置）、**有条件的执行**（*conditional execution*，如果布尔条件是 false，则向前跳到

高级实现

```
//while 循环
While (R1>=0) {
   代码段 1
}
代码段 2
```

底层实现

```
//while 循环结构的典型翻译;
beginWhile
   JNG R1, endWhile   //If R1<0 goto endWhile
   //这里放置代码段 1 的翻译内容
JMP beginWhile    //Goto beginWhile
endWhile:
   //这里放置代码段 2 的翻译内容
```

图 4.1 高级和底层分支逻辑（branch logic）。虽然不同语言的 *goto* commands 语法不同，但基本思想是一样的

“if-then” 语句之后的位置），以及**子程序调用**（*subroutine calling*，跳到另一代码段的第一条命令处）。为了支持这些程序结构，各种机器语言都可以有条件（conditional）或无条件（unconditional）地跳转到程序指定的地址。在汇编语言中，程序中的位置也用一些符号表示。图 4.1 给出了典型的例子。

无条件跳转（*unconditional jump*）命令（比如 `JMP beginWhile`）只指定了跳转的目标地址。有条件跳转命令比如 `JNG R1`、`endWhile` 还必需给出确定的布尔条件。在一些语言中，布尔条件可以通过命令显式地给出，也可以隐含地由先前指令的执行结果给出。

关于机器语言及其支持的通用操作，就简单介绍到这里。下一节针对一种特定的机器语言——即第 5 章构建的计算机所采用的机器语言——进行详细阐述。

4.2 Hack 机器语言规范详述
Hack Machine Language Specification

4.2.1 概述
Overview

Hack 是一个基于冯·诺伊曼架构的 16-位计算机，由一个 CPU、两个独立的内存模块（instruction memory 即指令内存和 data memory 即数据内存），以及两个内存映射 I/O 设备（显示器和键盘）组成。

内存地址空间（Memory Address Space） Hack 程序员要了解，有两个不同的地址空间：**指令地址空间**（*instruction memory*，以下称为**指令内存**），**数据地址空间**（*data memory*，以下称为**数据内存**）。两个内存区都是 16-位宽，有 15-位地址空间，这意味着两个内存可设的最大地址都是 32K 的 16-bit word。

CPU 仅能执行存储在指令内存中的程序。指令内存是只读设备，程序通过某种外部方法被加载到指令内存中。比如，可以在预先烧写了必要程序的 ROM 芯片中实现指令内存。要加载新程序，则要替换整个 ROM 芯片，跟玩游戏时需要在游戏控制台中替换游戏卡一样。为了模拟这个操作，Hack 平台的硬件仿真器必须提供一种方法，将某文本文件中用机器语言编写的程序加载到指令内存中去。

寄存器（Registers） Hack 程序员要接触两个称为 D 和 A 的 16-位寄存器。这些寄存器能够被算术和逻辑指令显式地操控，比如 `A=D-1` 或 `D=!A`（这里“`!`”表示 16-位的 Not 操作）。D 仅用来储存数据值，A 既可作为数据寄存器也可作为地址寄存器。也就是说，根据指令的上下文含义，A 中的内容可以被解释为数值或数据存储器中的地址，或者作为指令存储器中的地址。

首先，A 寄存器能够使“对数据内存（以后简称为‘内存’）的直接存取”变得十分容易。由于 Hack 的指令是 16-位宽，而对地址的描述要用到 15 位，所以将操作码和地址都存放在同一条指令中是不可能的。因此，Hack 语言的语法规定，内存的存取指令是对隐式的内存地址“M”进行操作，比如 `D = M + 1`。为了解析这个地址，规定 `M` 总是代表一个内存单元中的数值，该内存单元的地址就是当前 A 寄存器中的数值。比如说，如果想要执行操作 D=Memory[516]－1，就必须使用一条指令来将 A 寄存器的值置为 516，然后使用指令 `D = M - 1` 来完成操作。

另外，身兼重任的 A 寄存器也被用来对指令存储器进行直接访问。与内存访问规则一致，Hack 的 jump 指令并不指定某个特定地址。其规定是：任何 jump 操作总是执行这样的跳转，即跳转到“A 寄存器所指定的指令”。如此一来，若要执行操作 *goto* 35，就要使用一条指令将 A 的值置为 35，然后第二条指令就直接使用无需描述地址的 *goto* 命令。这个操作序列使计算机在下一个时钟周期获取在 InstructionMemory[35] 位置上的指令。

范例 因为 Hack 语言的语法是很容易理解的，我们就从一个例子开始说明。Hack 语言中唯一需要解释的命令就是@*value* ，这里 *value* 可以是数值或是代表数值的符号。这个命令将特定的值存到 A 寄存器中。比如，如果 `sum` 代表内存地址 17，那么`@17` 和`@sum` 都将具有一样的功能：`A ← 17` 。

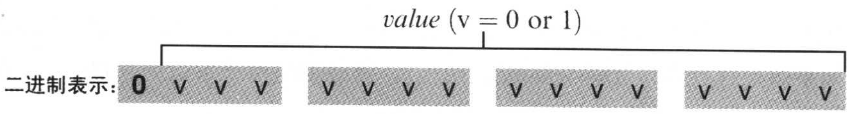
回到这个例子：假设我们想要进行从 1 到 100 的连加。图 4.2 给出了 C 语言的算法和 Hack 语言的汇编程序。

虽然 Hack 语法比大多数机器语言的语法更好理解，但对于不太熟悉低级程序语言的读者而言，它仍然很晦涩。特别需要注意的是，每个涉及内存地址的操作需要两个 Hack 命令：一个用来确定将要进行操作的内存单元地址，另一个用来描述要进行的操作。Hack 语言正包含了两种指令：一种是**地址指令**，也称为 A-指令；另一种是**计算指令**，也称为 C-指令。每种指令都有二进制表示法和符号表示法，并且都能对计算机产生特定的作用。

4.2.2 A-指令
The A-Instruction

A-指令用来为 A 寄存器设置 15-位的值：

A-instruction: @*value* // 这里 *value* 是一个非负的十进制数
// 或者一个代表非负十进制数的符号



这个指令使得计算机将特定的值储存到 A 寄存器中去。比如说，指令`@5`，也等价于 `0000000000000101`，则使得计算机将用二进制表示的 5 储存到 A 寄存器中。

A-指令主要有三种不同的用途。首先，在程序控制下，它提供了唯一一种“将常数输入计算机”的方法；其次，通过将目标数据内存单元的地址放入 A 寄存器，来为将对该内存单元进行操作的 C-指令提供必要的条件；其三，通过将跳转的目的地址放入 A 寄存器来为执行跳转的 C-指令提供条件。A-指令的用途可以参看图 4.2。

C 语言

```
// 1+...+100 的累加
   int i = 1;
   int sum = 0;
   While (i <= 100){
       sum += i;
       i++;
   }
```

Hack 机器语言

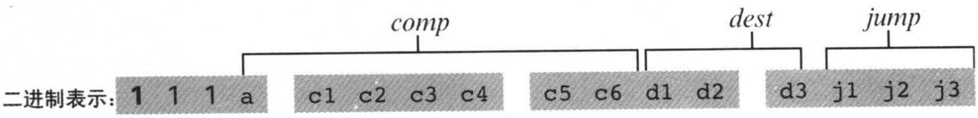
```
//Adds 1+...+100 的累加
        @i        // i 指代某个内存单元。
M=1               // i=1
@sum              // sum 指代某个内存单元。
M=0               // sum = 0
(LOOP)
@i
D=M               // D= i
@100
D=D-A             // D=i-100
@END
D;JGT             // If (i-100)>0 goto END
@i
D=M               // D=i
@sum
M=D+M             // sum=sum + i
@i
M=M+1             // i= i+1
@LOOP
0;JMP             // Goto LOOP
(END)
@END
0;JMP             //无限循环
```

图 4.2 同一程序的 C 语言和汇编语言版本。程序最后的无限循环是我们“结束”Hack 程序执行的标准方式

4.2.3 C-指令
The C-Instruction

C-指令是 Hack 平台程序的重点，几乎包含要做的所有事情。指令代码的描述可以说是对以下三种问题的回答：（a）计算什么；（b）将计算后的值存储到什么地方；（c）下一步做什么。随同 A-指令一起，这些描述几乎决定了计算机所有可能的操作。

```
C-instruction:  dest = comp;jump     // dest 或者 jump 可以为空
                                     // 如果 dest 为空，"=" 省略
                                     // 如果 jump 为空，";" 省略
```



指令编码最左边起第一位为 1，则表示该指令是 C-指令。接着的两位没有被使用。剩下的位构成了三个域，分别代表指令符号表述的三个部分。符号指令 *dest = comp;jump* 的整个语义可以如下表示。*Comp* 域告诉 ALU 计算什么。*Dest* 域指明计算后的结果（ALU 的输出）将被存储到什么地方。*Jump* 域描述了转移条件，即接下来要取出并执行哪一条命令。现在来介绍这三个域的格式和语法。

Computation 规范（计算规范） Hack 的设计中，ALU 所执行的是一组固定的函数集，该函数集的功能实现了对寄存器 D、A、M（M 代表 Memory[A]）的操作。计算函数指令的 *comp* 域由 1 个 a-位域（a-bits）和 6 个 c-位域（c-bits）组成。这个 7-位模式可以对 128 个不同的函数进行编码，但只有其中的 28 个函数（如图 4.3）在机器语言规范中列出。

前面介绍过 C-指令的格式是 `111a cccc ccdd djjj`。若希望让 ALU 计算 `D-1`，即 D 寄存器当前值减 1。根据图 4.3，可以通过指令 111**0 0011 10**00 0000（7 位操作码被加粗显示）来完成。要计算 `D|M` 的值，则通过指令 111**1 0101 01**00 0000 来完成。如果要计算常数－1，则利用指令 111**0 1110 10**00 0000 来完成。

Destination 规范（目的地规范） C-指令的 *comp* 部分计算出来的值能够被存储在不同

(当a=0) comp 助记符	c1	c2	c3	c4	c5	c6	(当a=0) comp 助记符
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

图 4.3 C-指令的 *compute* 域。D 和 A 是寄存器的名字。M 代表内存单元，该内存单元的地址由寄存器 A 给出，也就是 Memory[A]。符号+和-表示 16-位 2-补码的加法和减法，!、|和&分别表示 16-位的按位布尔操作 Not、Or 和 And。注意该指令集与图 2.6 中 ALU 规范的相似之处

的目的地址单元，而具体的位置是由指令的 3-位 *dest* 域（如图 4.4 所示）来描述的。第一位和第二位分别决定是否将计算结果存入 A 或 D。第三个 *d*-位域（*d*-bits）决定是否将值存入 M（比如 Memory[A]）中。

如果希望计算机将 Memory[7] 的值加上 1，并且将结果也存入 D 寄存器中。根据图 4.3 和图 4.4，可以通过如下的指令来完成：

```
0000 0000 0000 0111      //@7
1111 1101 1101 1000      //MD = M + 1
```

d1	d2	d3	助记符	目的地（存储计算后的值）
0	0	0	Null	代表“空”（NULL），该值不会被存储
0	0	1	M	Memory[A]（内存单元的地址由寄存器A给出）
0	1	0	D	D寄存器
0	1	1	MD	Memory[A] 和D寄存器
1	0	0	A	A寄存器
1	0	1	AM	A寄存器和Memory[A]
1	1	0	AD	A寄存器和D寄存器
1	1	1	AMD	A寄存器，Memory[A] 和D寄存器

图4.4 C-指令的 *dest* 域

第一条指令使得计算机选中地址为7的寄存器（也就是M）。第二条指令计算M+1的值，并且将结果存入M和D。

Jump规范（跳转规范） C-指令的 *jump* 域告诉计算机下一步将执行什么命令。有两个可能性：计算机获取并执行程序中紧接着当前指令的下一条指令，这是默认的情况；或者它获取并执行程序中位于其他地址的一条指令。对于后者我们假设A寄存器已经装载了跳转的目的地址。

Jump操作能否实际执行取决于 *jump* 域的三个 *j*-位域（*j*-bits）和ALU的输出值（根据 *comp* 域计算出来的）。第一个 *j* 位规定：当ALU输出值小于0时，发生跳转；第二个 *j* 位规定：当ALU输出值等于0时，发生跳转；第三个 *j* 位规定：当ALU输出值大于0时，发生跳转；这样共有8种可能的组合条件实现跳转。这样共有八种可能的跳转条件的组合，如图4.5所示。

下面给出了一个jump命令的例子：

逻辑

```
if Memory[3]=5 then goto 100
else goto 200
```

实现

```
@3
D=M    // D=Memory[3]
@5
D=D-A // D=D-5
@100
D;JEQ // If D=0 goto 100
@200
0;JMP // Goto 200
```

j1 ($out<0$)	j2 ($out=0$)	j3 ($out>0$)	助记符	作 用
0	0	0	null	No jump
0	0	1	JGT	If $out>0$ jump
0	1	0	JEQ	If $out=0$ jump
0	1	1	JGE	If $out\geq 0$ jump
1	0	0	JLT	If $out<0$ jump
1	0	1	JNE	If $out\neq 0$ jump
1	1	0	JLE	If $out\leq 0$ jump
1	1	1	JMP	Jump

图 4.5 C-指令的 *jump* 域。*Out* 指代 ALU 输出（由指令的 *comp* 部分计算得到），*jump* 意味着"跳转到由 A 寄存器确定的指令继续执行"

最后一个指令（`0;JMP`）执行一个无条件的跳转。因为 C-指令语法要求我们必须执行一些计算，所以干脆让 ALU 去计算 0（其实可取任意值），计算机会忽略它。

使用 A 寄存器的冲突 A 寄存器使用上的冲突正如刚才所讲，程序员可以使用 A 寄存器为"包含 M 的 C-指令"指定**数据内存**中的地址，也可以为"包含 jump 的 C-指令"指定**指令内存**中的地址。因此，为了避免 A 寄存器在使用上的冲突，在编写良好的程序中，在可能引发 jump（即有一些非零的 *j* 位）的 C-指令中不能引用 M，反之亦然（即在"引用 M 的 C-指令"中也不准引发 jump）。

4.2.4 符号
Symbols

汇编命令可以使用常数或符号来表示内存单元位置（地址）。通过以下三种方式应用到汇编语言中：

- **预定义符号**（*Predefined symbols*）：RAM 地址的一个特殊子集可以通过如下预定义符号来被所有汇编程序引用：

- **虚拟寄存器**（*Virtual registers*）：为了简化汇编程序的编写，用符号 `R0` 到 `R15` 用来代表 0 到 15 号 RAM 地址。

- **预定义指针**（*Predefined pointers*）：符号 `SP`、`LCL`、`ARG`、`THIS` 和 `THAT` 被预定义为表示 0 到 4 号 RAM 地址。注意这四个内存位置都有两种符号表示。

比如，address 2 可以使用 R2 或 ARG 来表示。这种表示方法会在第 7、8 章中讨论的虚拟机实现中使用到。

- I/O 指针：符号 SCREEN 和 KBD 被预定义以表示 RAM 地址 16384（0x4000）和 24576（0x6000），这两个地址分别是屏幕和键盘内存映像（memory-map）的基地址。这些 I/O 设备的使用会在后面介绍。

■ **标签符号**（*Label symbols*）：用户自定义符号用来标记 *goto* 命令跳转的目的地址。由伪指令“(Xxx)”来声明用户自定义的符号，其意义是：Xxx 代表程序中下一条命令的指令内存位置。一个标签只能被定义一次，可以在汇编程序中的任何地方使用，即使是在其定义之前也可以使用。

■ **变量符号**（*Variable symbols*）：在汇编程序中任何用户定义的符号 Xxx，如果它不是预定义符号，也不是使用“(Xxx)”的标签符号，那么就被看作是变量，并被汇编程序赋予独立的内存地址（从 RAM 地址 16，即 0x0010 开始）。

4.2.5 输入/输出处理 Input/Output Handling

Hack 平台能够连接两个外部设备：屏幕和键盘。两个设备与计算机平台的交互都是通过**内存映像**（*memory maps*）实现的。这意味着在屏幕上描绘像素是通过将二进制值写入与屏幕相关的内存段来实现。同样，键盘的输入是通过读取与键盘相关的内存单元来实现的。物理 I/O 设备和它们对应的内存映像是通过连续的循环刷新来同步的。

屏幕 Hack 计算机包括有一个黑白屏幕，256×512 像素。屏幕的内容由 RAM 基地址为 16384（0x4000）的 8K 内存映射来表示。物理屏幕的每一行从屏幕的左上角开始，在 RAM 中用 32 个连续的 16-位字表示。因此至顶部 *r* 行、至左边 *c* 列的像素映射到位置为 RAM[16394+*r*·32+*c*/16] 的字的 *c*%16 位（从 LSB 到 MSB）。为了读写屏幕上的一个像素，可以对 RAM 内存映射的相关位进行读写操作（1=黑，0=白）。比如：

```
//在屏幕的左上角画一个黑点
@SCREEN  //将 A 寄存器的值置为内存映射区的映射到屏幕第一行
         //的 16 个最左边的像素的内存字。
M = 1    //将最左边的像素变黑。
```

键盘 Hack 计算机与物理键盘之间通过 RAM 基地址为 24576（0x6000）的单字内存映像进行交互。只要在键盘上敲击一个键，其对应的 16-位 ASCII 码值就出现在 RAM[24576]；没有击键时，该内存单元的值就为 0。除了常用的 ASCII 码之外，Hack 键盘还可以识别图 4.6 中列出的一些键。

4.2.6 语法规约和文件格式
Syntax Conventions and File Format

二进制文件 二进制文件由文本行组成。每行都是一连串的 16 个“0”和“1”ASCII 字符，其对应一条单独的机器语言指令。文件中的所有行组成了机器语言程序。当机器语言程序被加载到计算机指令内存中，约定文件中第 *n* 行所表示的二进制码被存储到指令内存地址为 *n* 的单元中（程序行数和内存地址计数都是从 0 开始计）。按照惯例，机器语言程序被存进扩展名为“hack”的文本文件中，比如 Prog.hack。

汇编语言文件 习惯上，汇编语言程序存储在以“asm”为扩展名的文本文件里，比如 Prog.asm 。汇编语言文件由文本行组成，每一行代表一条**指令**或者一个**符号定义**：

按 键	编 码	按 键	编 码
newline	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
up arrow	131	insert	138
right arrow	132	delete	139
down arrow	133	esc	140
home	134	f1–f12	141–152

图 4.6 Hack 平台中特殊键所对应的编码

- **指令**（*Instruction*）：一条 A-指令或 C-指令
- *(Symbol)*：这条伪指令会让编译器把 `Symbol` 标签分配给程序中下一条命令被存储的内存单元地址。因为它不生成任何机器代码，所以称之为“伪命令(pseudo-command)”。

（下面的一些规约只适合于汇编程序。）

常数（constants）**和符号**（symbols）　常数必须是非负的而且总是用十进制表示。用户自定义的符号可以是任何字母、数字、下划线（_）、点（.）、美元符号（$）、冒号（:）构成的字符串，但它不能以数字开头。

注释　以双斜线（//）开头，并到本行结束符之前（不换行）的文本被认为是注释，不被程序执行。

空格　空格字符及空行也被程序忽略。

大小写　所有的汇编助记符必须大写。其他的比如用户自定义标签（label）和变量名则要区分大小写。一般是标签大写，变量名小写。

4.3 观点 Perspective

Hack 机器语言几乎是所有机器语言中最简单的一种。大多数计算机都有更多的指令、更多的数据类型、更多的寄存器、更多的指令格式、更多的寻址模式。然而，任何 Hack 机器语言不支持的功能都可以在软件中实现。比如，Hack 平台的原始机器语言操作不支持乘法和除法。显然这些操作在任何高级语言里面都是需要的，所以我们后面会在操作系统中去实现它们。

从语法上来看，我们尽量不让 Hack 同大多数汇编语言一样机械化。我们特别为 C-命令选择了一种类似于高级语言的语法，比如，使用 `D=M` 和 `D=D+M`，而不是传统的 LOAD 和 ADD 指令。读者要注意，这些仅仅是语法的细节。例如，在“`D=D+M`”命令中，“`+`” 字符

并不发挥代数上的运算作用。三个字符组成的字符串“`D+M`”是作为整体，被当成单独的汇编助记符来看待，对应于一个 ALU 操作的编码。

单一指令所能包含内存地址的数量反映了不同机器语言的各自特点。这样，Hack 可以被描述成一个“1/2 地址机器”。因为在 16-位指令格式中无法包含一个指令码和一个 15-位的地址，所以在 Hack 中，涉及内存访问的一些操作通常使用两个指令来描述：一个 A-指令来描述地址，一个 C-指令来描述操作。相比之下，大多数机器语言能够在每个机器指令中直接指定至少一个地址。

事实上，Hack 汇编代码的典型结构是 A-指令和 C-指令的连续交替，比如，`@xxx` 后面通常接着 `D=D+M` ，`@YYY` 后面通常接着 `0;JMP`，等等。如果你觉得这种编写方式太单调而且繁琐，可以在程序中使用一些更友好的**宏命令**（*macro commands*），比如 `D=D+M[xxx]` 和 `GOTO YYY` ，让 Hack 汇编代码更易阅读且只有原来代码的一半长度。其关键是要将这些宏命令翻译成二进制码（比如后跟 `D=D+M` 的`@xxx` ，后跟 `0;JMP` 的`@YYY` 等）来实现。

汇编编译器（*assembler*）在本章里面提到了很多次，它是一种程序，负责将以符号形式表达的汇编程序翻译成用二进制代码编写的可执行程序。除此之外，它还负责管理汇编程序中所有系统符号（system symbols）和用户定义符号（user-defined symbols），并在编译过程中以物理内存地址来替换它们。我们会在第 6 章中具体介绍这个翻译任务，在那一章里会为 Hack 语言构建汇编编译器。

4.4 项目 Project

目标 初步了解机器语言底层编程，熟悉 Hack 计算机平台。在做本项目的过程中，你会熟悉汇编过程，同时也能形象地理解翻译后的二进制码是如何在目标硬件上执行的。

资源 在这个项目里你将会用到本书配套软件包的两个工具：汇编编译器，用来将 Hack 汇编程序翻译成二进制码；以及 CPU 仿真器，用来在虚拟的 Hack 平台上运行二进制程序。

约定 编写并测试下面两个程序，你的程序在 CPU 仿真器上执行时，应该会经由项目路径中的测试脚本的领引，从而产生相应的结果。

■ **乘法程序**（Mult.asm）：该程序的输入值存储在 R0 和 R1 中（也就是内存的两个起始内存单元）。程序计算 R0*R1 的值并将其存入 R2。我们假设（此程序中）R0>=0，R1>=0，R0*R1<32768，虽然这些条件程序自身不用去检测，但是作为程序设计者的你必须保证它们成立。提供的 Mult.tst 和 Mult.cmp 脚本会用一些典型的数据值来测试你的程序。

■ **I/O 处理程序**（Fill.asm）：这个程序是一个无限循环，它侦测键盘的输入。当按下任一键时，程序将屏幕变黑，即将"black"写入每个像素。当没有键按下时，屏幕应该被清屏。你以任何空间顺序来选择屏幕的变黑和清屏，只要连续地按一个键足够长时间，屏幕就会全黑，长时间不按键就会清屏。这个程序有测试脚本（Fill.tst）但是没有比较文件（.cmp），只有通过对模拟屏幕的观察来检查执行结果。

步骤 推荐按如下步骤进行：

0. 该项目所需要的汇编编译器和 CPU 仿真器在本书配套软件包中的 tools 路径下。使用它们之前，请先阅读汇编编译器参考手册和 CPU 仿真器参考手册。

1. 使用普通的文本编辑器编写第一个汇编程序，将它保存为 projects/04/mult/Mult.asm。

2. 使用提供的汇编编译器（以批处理或者交互模式）来翻译你的程序。如果程序中有语法错误，回到步骤 1。如果没有语法错误，编译器将产生名为 projects/04/mult/Mult.hack 的包含二进制机器指令的文件。

3. 使用提供的 CPU 仿真器来测试生成的 Mult.hack 代码。测试过程可以使用 Mult.tst 脚本以交互方式或者批处理方式来实现。如果产生了运行期错误，返回到步骤 1 检查。

4. 对第二个程序（Fill.asm）重复步骤 1～3，路径为 projects/04/fill。

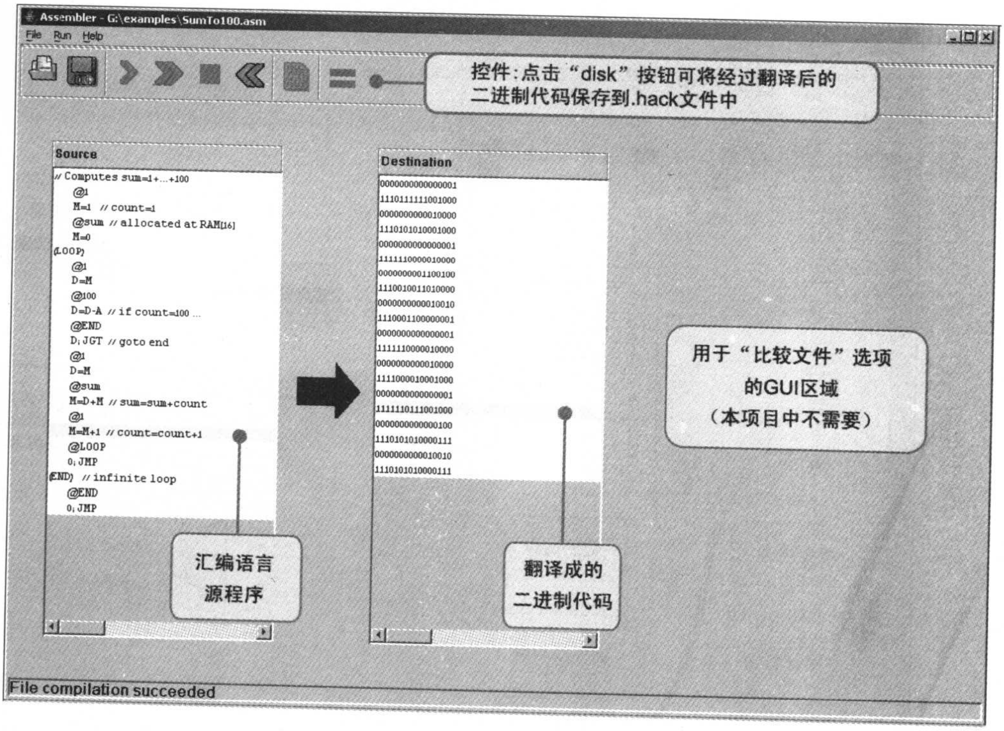


图 4.7 本书提供的可视化汇编编译器

调试提示 Hack 语言是区分大小写的。程序员在编写程序时常犯的错误是认为`@foo`和`@Foo`所代表的是同一个变量。事实上，编译器将这些符号看作是两个完全不同的标识符（identifiers）。

汇编编译器 与本书配套的软件包所包含Hack汇编编译器，可以在命令模式或者GUI模式下使用。后一种操作模式以可视化和单步执行的方式来观察编译过程，如图 4.7 所示。

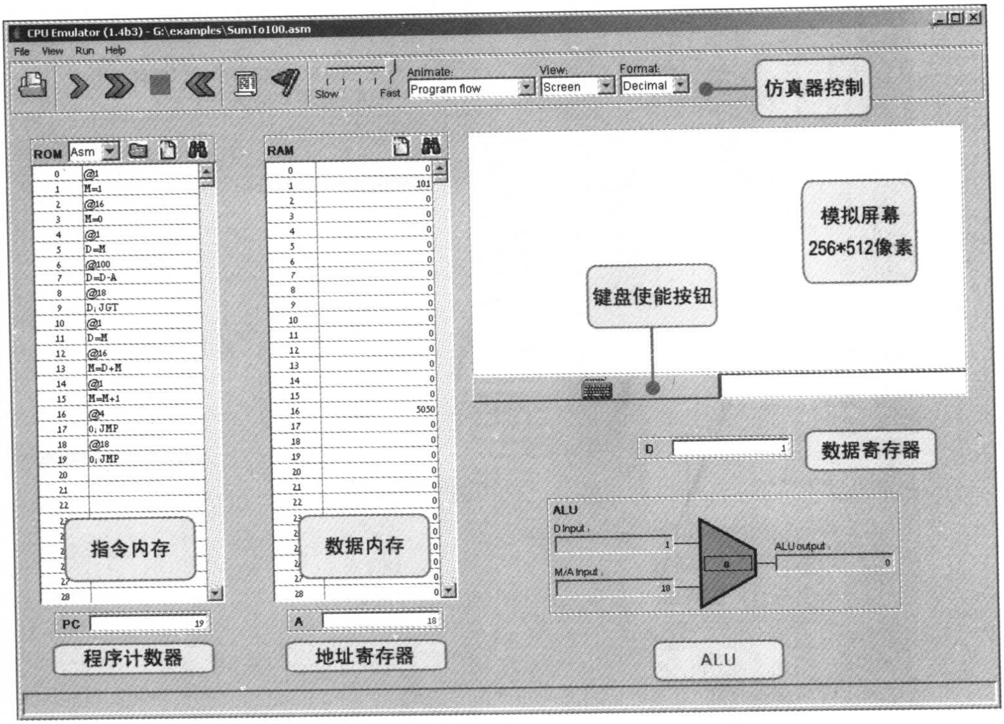


图 4.8 本书提供的 CPU 仿真器。加载的程序可以是符号化的方式表示（如该屏幕截图所示）或者以二进制码的方式表示。该程序中并没有使用屏幕和键盘

对汇编编译器产生的机器语言程序可以采用两种不同的方式来进行测试。我们可以在 CPU 仿真器中运行 `.hack` 程序，或者使用硬件仿真器将程序加载到计算机的指令内存中，直接在硬件上运行相同的程序。因为在下一章才会结束硬件平台的构建，所以目前为止前一种方式更有意义。

CPU 仿真器 该程序模拟 Hack 计算机平台。它允许将 Hack 程序加载到模拟的 ROM 中，并且能够观察它在模拟的硬件上执行的过程，如图 4.8 所示。

为了方便使用，CPU 仿真器能够加载二进制`.hack`文件和`.asm`的汇编文件。对于`.asm`文件，仿真器即时地将汇编程序翻译成二进制代码。看上去编译器像是多余的，但是事实并非如此。首先，编译器能够显示翻译过程。其次，编译器产生的二进制文件能够直接在硬件平台上执行。为了做到这点，将（第 5 章中项目构建的）计算机芯片加载到硬件仿真器中，然后将编译器产生的`.hack`文件加载到计算机的 ROM 芯片中。

第 5 章　计算机体系结构

Computer Architecture

Form ever follows function.
形式永远跟随功能。
——Louis Sullivan（1856 ~ 1924），现代主义建筑师
Form IS function.
形式就是功能。
——Ludwig Mies van der Rohe（1886 ~ 1969），现代主义建筑师

本章涵盖全书“硬件”部分当中最难啃的内容，将在第 1 至 3 章中构建的所有芯片整合起来，集成为一台通用计算机，使之能够运行用机器语言编写的程序。将要构建的计算机称为 Hack，Hack 有两个很重要的优点：一方面，Hack 是简单的计算机，通过使用前面构建的芯片和与本书配套的硬件仿真器，Hack 能够在几个小时内被构建起来。另一方面，Hack 计算机的体系结构足以描述任何数字计算机的关键操作原理和硬件组成。因此，构建 Hack 将会使读者对现代计算机如何在较底层的硬件和软件上运行有很好的理解。

5.1 节首先介绍了**存储程序**（*stored program*）的概念，然后对**冯·诺依曼结构**（*von Neumann architecture*，计算机领域里几乎所有现代计算机设计的中心教条）进行了详细地描述。Hack 平台是冯・诺依曼机的实例，5.2 节详细介绍了其硬件结构。5.3 节描述了 Hack 平台是如何由一些芯片实现的，这些芯片包括第 2 章中构建的 ALU、第 3 章中构建的寄存器[1]和内存系统等。

基于此结构构建出来的计算机，是尽可能简单而又不至于简单过度。这意味着它会具有最小系统的配置以运行程序并体现合理的性能。5.4 节中会将这种计算机与典型计算机进行比较，对比着重于体现“**优化**（*optimization*）”在工业级计算机设计中的关键角色，

[1] 请参见第 41 页关于“寄存器”概念的脚注。——审校者

但跟前面介绍的一样，方法简单性的目的在于：所有本章中提到的用于构建 Hack 计算机的芯片——甚至是 Hack 计算机本身——能够在一台运行硬件仿真器的个人计算机上进行构建和测试（这会在 5.5 节中介绍）。最后将生成一台具有最小规模，但却拥有惊人强大功能的计算机。

5.1 背景知识 Background

5.1.1 存储程序概念 The Stored Program Concept

与我们周围的所有其他机器相比，数字计算机最独特的特点在于其功能多样性。一个由有限硬件组件构成的计算机却可以执行无限的任务队列，从交互式游戏到字处理到科学计算。这种卓越的灵活性看似理所当然，而其实这都是“**存储程序**（*stored program*）”概念的功劳硕果。这个概念在 20 世纪 30 年代曾被几位著名数学家形式化描述过，到现在它仍然被认为是现代计算机史上最具意义的发明（即使不能算是最基础的发明）。

跟许多科学上的突破一样，存储程序概念的基本思想其实相当简单。计算机基于固定的硬件平台，能够执行固定的指令集。同时，这些指令能够被当成构件模块，组成任意的程序。而且，不同于 1930 年以前的机械计算机，这些程序的逻辑并没有被嵌入到硬件中，而是被存储到计算机的存储设备（memory）里，**跟数据一样**，成为所谓的“**软件**（*software*）”。因为计算机的操作是通过当前正在执行的软件来向用户展示其功能，所以每次向计算机中载入不同的程序时，同样的硬件平台可以实现完全不同的功能。

5.1.2 冯·诺依曼结构 The von Neumann Architecture

存储程序概念是很多抽象的、应用型计算机模型的关键，其中最著名的是**通用图灵机**（1936）和**冯·诺依曼机**（1945）。图灵机是描述虚拟的简单计算机的抽象机，主要用来分

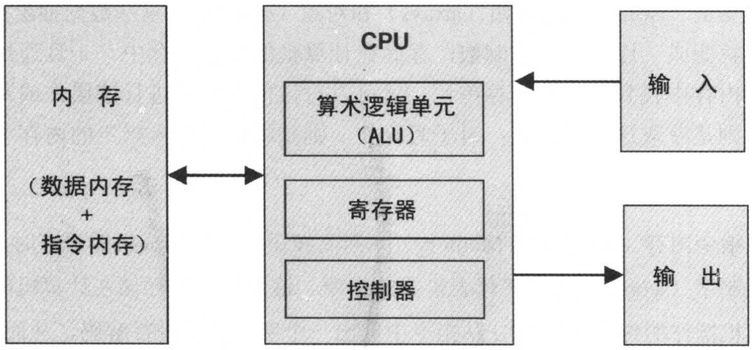


图 5.1 冯·诺依曼体系结构（概念上的）。在这个概念层上，该模型几乎可以描述所有数字计算机的体系结构。操作计算机的程序驻留在内存中，这与“存储程序（stored program）”的概念是一致的

析计算机系统的逻辑基础。相比之下，冯·诺依曼机是实际应用型的体系结构，它几乎是今天所有计算机平台的基础。

冯·诺依曼体系结构的基础是一个**中央处理单元**（CPU，*Central Processing Unit*），它与**记忆设备**（*memory device*）即**内存**[2]进行交互，负责从**输入设备**（*input device*）接收数据，向**输出设备**（*output device*）发送数据（见图 5.1）。这个体系结构的核心是存储程序的概念：计算机内存不仅存储着要进行操作的数据，还存储着指示计算机运行的指令。下面让我们来详细了解这个体系结构。

5.1.3 内存
Memory

冯·诺依曼机的内存中存有两种类型的信息：数据项（data items）和程序指令（programming instructions）。对这两种信息通常采用不同的方式来处理，在某些计算机里，它们被分被别存储到不同的内存区中。尽管它们具有不同功能，但两种信息都以二进制数形式存储在具有通用结构的随机存储器中（一个连续的固定宽度的单元阵列，也称为字即word，或者存储单元，每个单元都有一个独立的地址）表示。因此，一个独立的字（代表一个数据项或者一条指令）通过它的地址来指定。

数据内存（Data Memory） 高级程序（high-level programs）操纵抽象的元件（artifacts），

[2] 冯·诺依曼体系所描述的计算机体系结构是一种抽象概念，这里所说的内存（memory）概念实际上是借用自研究大脑时所采用的“记忆（memory）”概念，比我们平时理解和使用的普通内存概念更为广泛和抽象。虽然我们这里采“内存”这个术语来进行阐述，敬请读者明晰：这里所讲的抽象“内存”概念含义更为广泛，代表任何具有存储功能的设备或组件，应该理解为“记忆设备（memory device）”。具体可参考冯·诺依曼的相关著作。——审校者

例如变量（variables）、数组（arrays）和对象（objects）。这些数据抽象被翻译成机器语言后，就变成一连串的二进制数，存储在计算机的数据内存中。一旦通过指定的地址，在数据内存中找到对应的内存单元，就可以对该内存单元进行**读操作**或**写操作**。对于读操作，则是获取这个字的值；对于写操作，则是将新值存入指定的内存单元中，取代原来的旧值。

指令内存（Instruction Memory）　当高级命令被翻译成机器语言时，它变成一系列的二进制字（word），这些字代表机器的指令。这些指令被存储在计算机的指令内存中。在计算机操作的每一步，CPU 从指令中取出一个字，对其进行解码，从而执行指定的指令，然后计算下一条将要执行的指令。因此，改变指令中的内容会完全改变计算机的操作。

驻留在指令内存中的指令格式遵守机器语言的规约。在一些计算机中，每个操作的规范及其代表其操作数的代码是用一个单一字长的指令来表示的。其他计算机则将这个规范分为几个单字来表示。

5.1.4　中央处理器
Central Processing Unit

CPU 是计算机体系的核心，负责执行已被加载到指令内存中的指令。这些指令告诉 CPU 去执行不同的计算，对内存进行读/写操作，以及根据条件跳转去执行程序中其他指令。CPU 通过使用三个主要的硬件要素来执行这些任务：**算术逻辑单元**（ALU，*Arithmetic-Logic Unit*），一组**寄存器**（*registers*）和**控制单元**（*control unit*）。

算术逻辑单元（ALU）　ALU 负责执行计算机中所有底层的算术操作和逻辑操作。比如，典型的 ALU 可以执行的操作包括：将两个数相加；检查一个数是否为整数；在一个数据字（word）中进行位操作；等等。

寄存器（Registers）　CPU 的设计是为了能够**快速**地执行简单计算。为了提高它的性能，将这些和运算相关的数据暂存到某个局部存储器中是十分必要的，这远比从内存中搬进搬出要好。因此，每个 CPU 都配有一组**高速寄存器**，每个寄存器都能保存一个单独的字。

控制单元（Control Unit）　计算机指令用二进制代码来表示，通常具有 16、32 或 64 位宽。在指令能够被执行之前，须对其进行解码，指令里面包含的信息向不同的硬件设备

（ALU，寄存器，内存）发送信号，指使它们如何执行指令。指令的解码过程是通过某些**控制单元**（*control unit*）完成的。这些控制单元还负责决定下一步需要取出和执行哪一条指令。

CPU 操作现在可以被描述成一个重复的循环：从内存中取一条指令（字）；将其解码；执行该指令，取下一条指令；如此反复循环。指令的执行过程可能包含下面的一些子任务：让 ALU 计算一些值，控制内部寄存器，从存储设备中读取一个字，或向存储设备中写入一个字。在执行这些任务的过程中，CPU 也会计算出下一步该读取并执行哪一条指令。

5.1.5 寄存器 Registers

内存访问是很慢的过程。当 CPU 被指示去取内存中地址 j 的内容时，下面的一些过程会连续发生：（a） j 从 CPU 传到 RAM；（b）RAM 的直接访问逻辑（direct-access logic）选中地址为 j 的寄存器；（c）RAM[j]的内容传回到 CPU。寄存器也能提供同样的数据访问功能，但没有来回的数据传递和寻址开销。首先，寄存器位于 CPU 芯片内部，所以对它们的访问几乎可以瞬间完成；其次，与数百万个内存单元相比，寄存器数量非常少。因此，机器语言指令可以使用短短的几个位就能指定要操作的寄存器在什么位置，这样指令格式也更短。

基于不同的目的，不同的 CPU 采用不同数量、不同类型的寄存器。在一些计算机体系结构中，每个寄存器可以有多种用途。

数据寄存器（Data registers） 这些寄存器为 CPU 提供短期记忆（memory）服务。比如，当计算$(a-b)\cdot c$时，必须首先计算$(a-b)$的值并记住它。虽然这个结果可以暂时地被存储到存储单元中，但更好的办法是存储在 CPU 内部即数据寄存器中。

寻址寄存器（Addressing registers） 为了进行读写，CPU 必须连续访问内存中的数据。这样我们必须确定被访问的内存字（word）所在的内存地址。在某些情况下这个地址作为当前指令的一个部分给出，而其他某些情况下它依赖于前面一条指令的执行结果。对于后者，这个地址应该被存储到某个寄存器中，使得该寄存器的内容在今后的操作中能够被当作存储单元的地址——这就需要用到**寻址寄存器**。

程序计数寄存器（Program counter register）　执行程序时，CPU 必须总是知道下一条指令在指令内存中的地址。这个地址保存在一个特殊的寄存器即**程序计数器**中（或称 PC，*Program Counter*）。PC 的内容就被当作从指令内存中取指令的地址。因此，在执行当前指令的过程中，CPU 通过两种方式之一来更新 PC 的内容：1）如果当前指令不包括 *goto* 命令，PC 增 1 以便使指针指向程序中的下一条指令；2）如果当前指令中包含需要执行的 *goto n* 命令，则 CPU 将 PC 置为 *n*。

5.1.6 输入和输出
Input and Output

计算机使用一组输入输出（I/O）设备来与其外部环境进行交互。这些设备包括屏幕、键盘、扫描仪、网络接口卡、光驱等，还有一些复杂的集成在汽车、武器系统、医疗器械等机器中的嵌入式计算机。这里我们不考虑这些设备本身的构造，主要是因为：首先，每个设备代表一块独立的机器，需要相关的工程知识；其次，计算机科学家设计不同的方案将这些不同外设的物理细节封装起来，让计算机能以相同的方式对它们进行操作，其中最简单的实现技巧之一就是 **I/O 映像**（*memory-mapped I/O*）。

I/O 映像的基本思想是：创建 I/O 设备的二进制仿真，使其对于 CPU 而言，“看上去”就像普通的内存段。特别地，每个 I/O 设备在内存中都分配了独立的区域，作为它的“内存映像”。对于**输入**设备（键盘、鼠标等），内存映像能连续不断地**反映**设备的物理状态；对于**输出**设备（屏幕、扬声器等），内存映像连续地**驱动**设备的物理状态。当外部事件作用于输入设备时（比如，在键盘上按键或者移动鼠标），某些特定的值就被写入它们各自对应的内存映像中。同样地，想要控制某些输出设备（比如，在屏幕上画图或者播放一首歌曲），就将一些特定值写入它们各自对应的内存映像。从硬件的角度来看，这个方案需要所有 I/O 设备提供类似于记忆单元（memory unit，或称内存单元）的那种接口。从软件的角度来看，需要对每个 I/O 设备定义交互协议，这样程序才能正确地访问它。若有大量可用的计算机平台和 I/O 设备，就不难明白各类**标准规范**（*standards*）在计算机体系结构设计中所起的重要作用。

I/O 内存映像体系的实际应用是很有意义的：CPU 以及整个平台的设计可以完全不依赖于要与计算机进行交互的 I/O 设备，也不依赖于 I/O 设备的数量和种类。无论何时将新的 I/O 设备与计算机连接，所要做的只是为其分配一个新的内存映像并记录它的基地址（这些配置工作是由操作系统来完成的）。这样一来，程序可以通过操控 I/O 内存映像中的比特数据来实现对相应物理 I/O 外设的操作。

5.2 Hack 硬件平台规范详述 The Hack Hardware Platform Specification

5.2.1 概述 Overview

Hack 平台是 16-位冯·诺依曼机，包括一个 CPU、两个独立的内存模块（指令内存和数据内存）和两个内存映像 I/O 设备（屏幕和键盘）。这个体系结构中的某些部分——尤其是它的机器语言——曾在第 4 章中介绍过。为了便于参考，这里给出了这些内容的概要。

Hack 计算机执行位于指令内存中的程序。指令内存是只读设备，因此必须使用一些外部方法将程序加载进去。比如，指令内存可以用 ROM 芯片来实现，该芯片预先烧写了所需要的程序。加载新的程序意味着要替换整个 ROM 芯片。为了模拟这个操作，Hack 平台的硬件仿真器提供了加载文本文件的方法，该文本文件包含用 Hack 机器语言编写的程序（从现在起，我们分别用 RAM 和 ROM 来指代 Hack 的数据内存和指令内存）。

Hack 的 CPU 由第 2 章介绍的 ALU 和三个分别称为**数据寄存器**（D，*data register*）、**地址寄存器**（A，*address register*）、**程序计数器**（PC，*program counter*）的寄存器组成。D 和 A 是通用的 16-位寄存器，它们可以被算术和逻辑指令（比如 `A=D-1`，`D=D|A` 等）使用。D 寄存器仅仅用来存储数据值；而对于 A 寄存器，根据指令的内容，A 寄存器中的内容可以被解释为数据值、RAM 地址或 ROM 地址。

Hack 机器语言有两种 16-位命令类型。其中地址指令的格式为 0vvvvvvvvvvvvvvv，这里 v 代表 0 或 1。这个指令使得计算机将 15-位常数 vvv...v 加载到 A 寄存器中。计算指令的格式则为 111accccccdddjjj 。根据 Hack 机器语言规范，a-位域（a-bits）和 c-位域（c-bits）指示 ALU 计算哪个函数，d-位域（d-bits）表示将 ALU 的输出存于何处，j-位域（j-bits）指定了可选的跳转条件。

计算机体系结构以这样的方式来进行连接：PC（程序计数器）芯片的输出端被连接到 ROM 芯片的地址输入端。如此一来，ROM 芯片总是输出 ROM[PC]（大小为一个 word），即“PC 所指向的指令内存单元”的地址。这个值称为**当前指令**（*current instruction*）。那么，每个时钟周期内整个计算机的操作可以表示为：

执行（*Execute*）：当前指令中不同的 bit 位域被同时被送入计算机中不同的芯片。如果其是**地址指令**（即 MSB=0），则 A 寄存器被置为指令中内含的 15-位常数。如果其是**计算指令**（即 MSB=1），则指令中内含的 a-位域、c-位域、d-位域和 j-位域被当作是控制位，导致 ALU 和寄存器会相应的去执行该指令。

取指令（Fetch）：计算机下一步取哪一条指令，取决于当前指令的 jump 位和 ALU 的输出。这些值共同决定了是否去执行跳转。如果是要执行跳转，则 PC 被置为 A 寄存器的值；否则就将 PC 的值加 1。在下一个时钟周期，PC 指向的指令在 ROM 的输出中出现，如此不断循环。

这个“取指令-执行”循环意味着在 Hack 平台里，涉及内存单元访问的基本操作通常需要两个指令：一个**地址指令**（*address instruction*），用于将给定的地址赋予 A 寄存器；以及一个后续的**计算指令**（*compute instruction*），用于对与该地址对应的内存单元进行操作（可以是对 RAM 单元的读/写操作，或是到 ROM 的跳转操作）。

现在开始详细介绍 Hack 硬件平台。开始之前还要指出，该平台可以由前面构建的一些部分来实现。CPU 是基于第 2 章构建的 ALU 的。**寄存器**和**程序计数器**就是第 3 章中介绍的 16-位寄存器和 16-位计数器。同样，ROM 和 RAM 芯片也是第 3 章中构建的内存单元的版本。最后，屏幕和键盘设备将通过内存映像与硬件平台进行接口，通过有相同接口的内置芯片（比如 RAM 芯片）来实现。

5.2.2 中央处理器
Central Processing Unit (CPU)

Hack 平台的 CPU 被设计用来执行 16-位指令，它与两个相互独立的内存模块相连：一个是指令内存，CPU 从该内存取指令；另一个是数据内存，CPU 可以对其进行读/写数据值，如图 5.2 所示。

5.2.3 指令内存
Instruction Memory

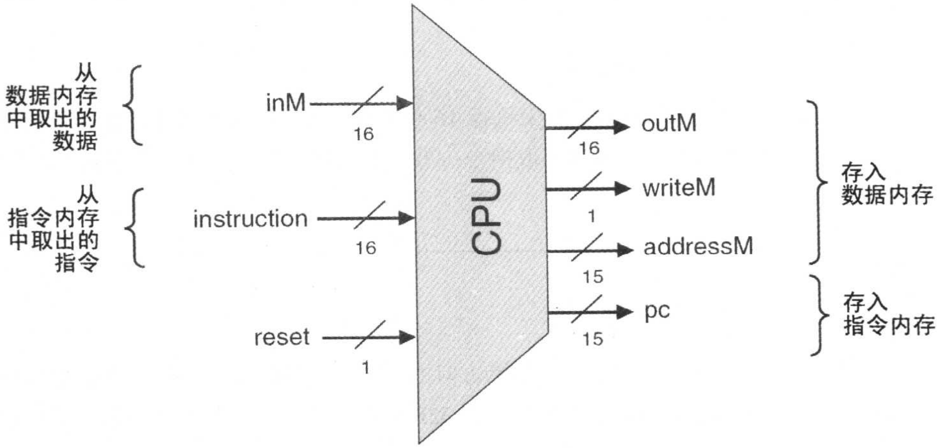
Hack 指令内存是可以进行直接访问的只读内存设备，也称为 ROM（Read-Only Memory）。Hack 的只读内存（ROM）由 32K 可寻址的 16-位寄存器组成，如图 5.3 所示。

5.2.4 数据内存
Data Memory

Hack 的**数据内存**芯片具有典型的 RAM 设备（跟第 3 章中构建的一样，请参看图 3.3）的接口。为了读取地址为 *n* 的内存单元中所存储的内容，就先将 *n* 放入内存的地址输入端口，然后从内存的输出端口得到结果，这是一个与时序无关的组合操作。为了将值 *v* 写入内存单元，则先将 *v* 置于内存的数据输入端口，然后将地址 *n* 置于地址输入端口，并且将内存的 `load` 比特位置为 1，这是一个时序操作，所以内存单元中的内容将在下个时钟周期变成 *n*。

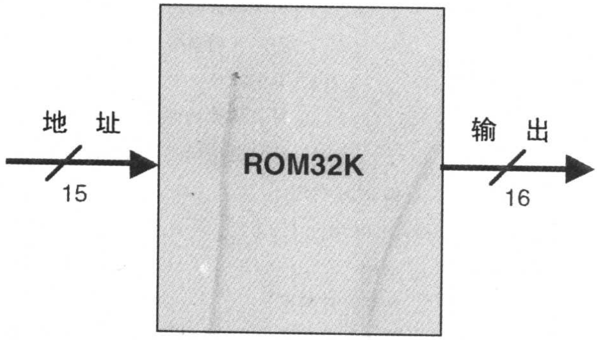
除了能够作为计算机的通用数据存储之外，数据内存还可以通过内存映像在 CPU 和计算机输入/输出设备之间充当接口。

内存映像（Memory Maps） 为了使与用户之间的交互变得简单，Hack 平台与两个外部设备相连：**屏幕**（*screen*）和**键盘**（*keyboard*）。两个设备都通过内存映像与计算机平台进行接口。通过对屏幕内存映像区段中的单元进行读写，分别实现得到屏幕状态和屏幕绘图操作。同样，可以通过检查键盘内存映像中的内存单元来得知当前按下了哪个键。内存映像通过存在于计算机之外的外部逻辑，与它们各自的 I/O 设备进行交互。交互协议如下：任何时候改变屏幕内存映像中的比特位，相对应的像素就被画在物理显示器上。任何时候在物理键盘上按下一个键，这个键值对应的编码将被存储到键盘内存映像中对应的内存单元内。




```
芯片名: CPU                         //中央处理器
输  入: inM[16],                    //输入值 M (M = RAM[A])
        instruction[16],            //用于执行的指令
        reset                       //决定是重启当前程序(reset=1)还是
                                    //继续执行当前程序(reset=0)的 rest 信号
输  出: outM[16],                   //输出值 M
        writeM,                     //写入 M ?
        addressM[15],               //M 在数据内存中的地址
        pc[15]                      //下一条指令的地址
功  能: 根据 Hack 机器语言规范执行指令。该语言规范指定 D 和 A 为 CPU 中的寄存器，M 代表内存单
        元，其地址由寄存器 A 中的内容确定(该内存单元中的数值从 inM 端输出)。
        如果指令希望将一个数值写入 M 中，该数值会从 outM 端输出，同时 M 的地址被置于 addressM
        端，而且 writeM 位被置位。(当 writeM=0，任何值都可能出现在 outM 。)
        如果 reset=1，那么 CPU 跳转到地址 0(也就是，在下一个时间单元设置 pc=0)，而不是执行
        当前指令所确定的下一条指令。
```

图 5.2　中央处理器。由第 2 章介绍的 ALU 和第 3 章介绍的寄存器组成



芯片名: ROM32K //容量为 32K 的 16-位只读内存
输　入: address[15] //ROM 中的地址
输　出: out[16] //ROM[address]中存储的数值
功　能: out=ROM[address] //16-bit 数值的赋值
说　明: 用机器语言编写的程序被预先载入 ROM 中。硬件实现可以将 ROM 当作一个内置芯片，而软件模拟器必须提供一种将程序载入 ROM 的机制。

图 5.3 指令内存

我们首先介绍在硬件接口和 I/O 设备之间相互作用的内置芯片，然后介绍嵌入了这些芯片的完整内存模块。

屏幕 Hack 计算机能够与一个黑白屏幕进行交互，该屏幕可以显示 256 像素行，每行宽 512 像素。计算机通过内存映像与物理显示设备（即屏幕）进行接口。也就是说，内存映像中的内容与屏幕上的内容是对应的，可以通过读写内存映像来操控物理显示设备（即屏幕）上的显示。内存单元中的数值对应于屏幕上的一个像素（1=黑色，0=白色）。内存映像和屏幕之间准确的映射如图 5.4 所示。

键盘 Hack 计算机跟个人计算机一样可以与标准键盘进行交互。计算机与物理键盘之间通过名为 Keyboard 的芯片进行交互（如图 5.5 所示）。当在物理键盘上按下一个键时，它的 16-位 ASCII 码作为 Keyboard 芯片的输出。当没有按下键时，芯片输出 0。除了通常的 ASCII 码之外，Keyboard 芯片还能识别并反映图 5.6 中所列的一些键。

```
芯片名: Screen                        //物理屏幕的内存映像
输  入: in[16],                       //要写入的内容
        load,                         //写使能位
        address[13]                   //写入的目的内存单元
输  出: out[16]                       //给定地址的屏幕值
功  能: 功能与一个 8K 的 16-位 RAM 相似:
        1. out(t)=Screen[address(t)](t)
        2. If load(t-1) then Screen[address(t-1)](t)=in(t-1)
        (t 是当前时间单元,或者时钟周期)
说  明: 模拟 256*512 黑白屏幕的持续刷新效果(仿真器必须模拟该设备)。从屏幕的左上角开始,物
        理屏幕上的每一行用 32 个连续的 16-位字表示。因此,顶起 r 行左起 c 列(0<=r<=255,
        0<=c<=511)的点与 Screen[r*32+c/16]代表的字第 c%16 位(从 LSB 到 MSB 换算)相
        对应。
```

图 5.4 屏幕接口

```
芯片名: Keyboard                      //物理键盘的内存映像
                                      //输出当前按下键的代码
输  出: out[16]                       //输出按键的 ASCII 码,
                                      //或者图 5.6 中列出的特殊码,
                                      //或者 0(没有任何键按下)
功  能: 输出当前在物理键盘上按下的键的代码。
说  明: 该芯片被物理键盘单元持续地刷新(仿真器必须模拟这个服务)。
```

图 5.5 键盘接口

按 键	键盘输出	按 键	键盘输出
newline	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
up arrow	131	insert	138
right arrow	132	delete	139
down arrow	133	esc	140
home	134	f1–f12	141–152

图 5.6 Hack 平台中的特殊键盘键

上面介绍了数据内存的内部单元，现在来介绍整个数据内存地址空间。

全局内存（overall memory）[3] Hack 平台的地址空间（也就是它的整个数据内存）是由一个称为 Memory 的芯片提供的。该芯片包含 RAM（用于常规的数据内存和指令内存）和屏幕、键盘内存映像。整个地址空间被分割成 4 个相互独立的子空间，数据内存、指令内存、键盘和屏幕各自占用一个子空间，如图 5.7 所示。

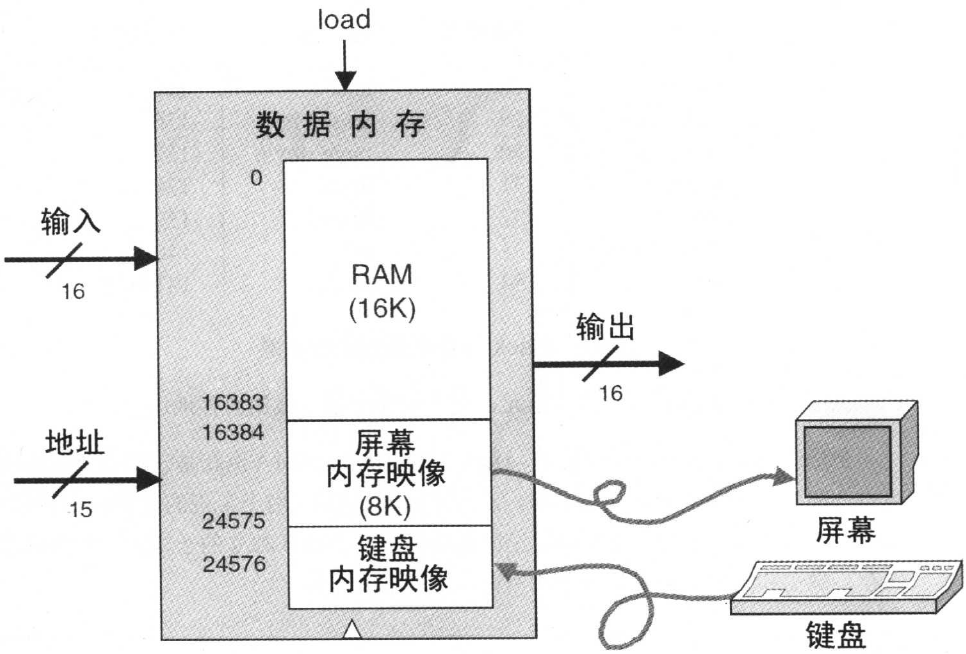
5.2.5 计算机 Computer

Hack 硬件阶层体系的顶层是一个单一的芯片（Computer 芯片），作为完整计算机系统的抽象，它能够执行用 Hack 机器语言编写的程序。图 5.8 展示了这种抽象。Computer 芯片包含了运行计算机所需要的所有硬件设备，包括 CPU、内存、键盘、屏幕，它们都被集成到该芯片内部，以实现整个计算机功能。为了执行一个程序，该程序的代码首先必须预加载到 ROM 中。对屏幕和键盘的控制则是通过它们的内存映像来完成的。

5.3 实现 Implementation

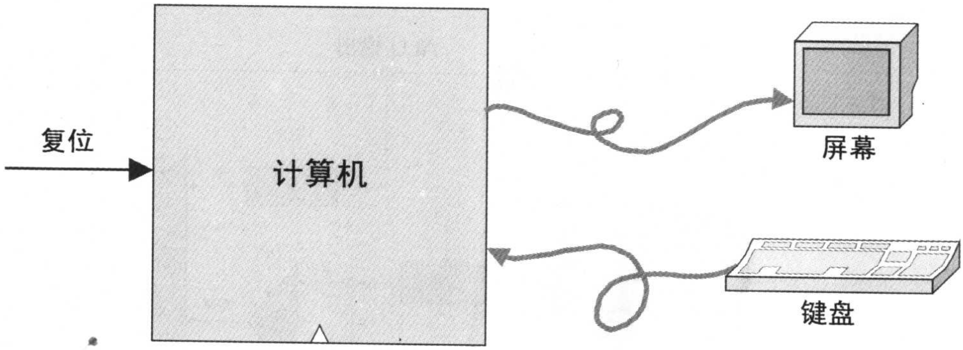
这一节主要介绍 Hack 计算机平台是如何被构建，从而实现规范中所描述的各种功能的。跟前面的章节一样，我们不给出详细的构建指令，而是希望读者自行设计。所有的芯片都可以用 HDL 语言来构建，并使用硬件仿真器在个人计算机上进行模拟。具体的技术细节会在本章最后的**项目**部分给出。

[3] 这里指的是从全局、整体的角度来考察整个内存区域。——审校者



芯片名:	Memory	//完整的内存地址空间
输　入:	in[16],	//要写入的内容
	load,	//写使能位
	address[15]	//要写入的目的内存单元
输　出:	out[16]	//指定地址所对应的内存单元中的值
功　能:	1. out(t)=Memory[address(t)](t) 2. If load(t-1) then Memory[address(t-1)](t)=in(t-1) (t 是当前时间单元，或者时钟周期)	
说　明:	对任何地址>24576 (0x6000) 的内存单元的访问是非法的。对地址 16384-24575 (0x4000-0x5FFF)范围的内存访问就是对屏幕内存映像的访问。对地址 24576 (0x6000) 内存单元的访问就是对键盘内存映像的访问。Screen 和 Keyboard 芯片规范描述了对这些内存映像的操作行为。	

图 5.7　数据内存



芯片名：Computer //最高抽象级的 Hack 芯片
输　入：reset
功　能：当 reset=0 时，计算机执行存储在 ROM 中的程序。当 reset=1 时，程序被重启。所以，要执行一个程序，首先应先将 reset 置为 1，然后再置为 0。
从这个角度观之，用户其实受惠于软件。根据程序的代码，屏幕显示相应的输出，用户可以通过键盘与计算机实现交互。

图 5.8 Computer，最高抽象级的 Hack“芯片”

因为 Hack 平台中的大部分行为都发生在它的中央处理器中，所以我们的主要任务就是构建 CPU。相比之下，计算机平台的其他部分的实现就很简单了。

5.3.1 中央处理器 The Central Processing Unit

CPU 的实现目标是建立逻辑门结构，其能够执行指定的 Hack 指令和读取下一条要执行的指令。

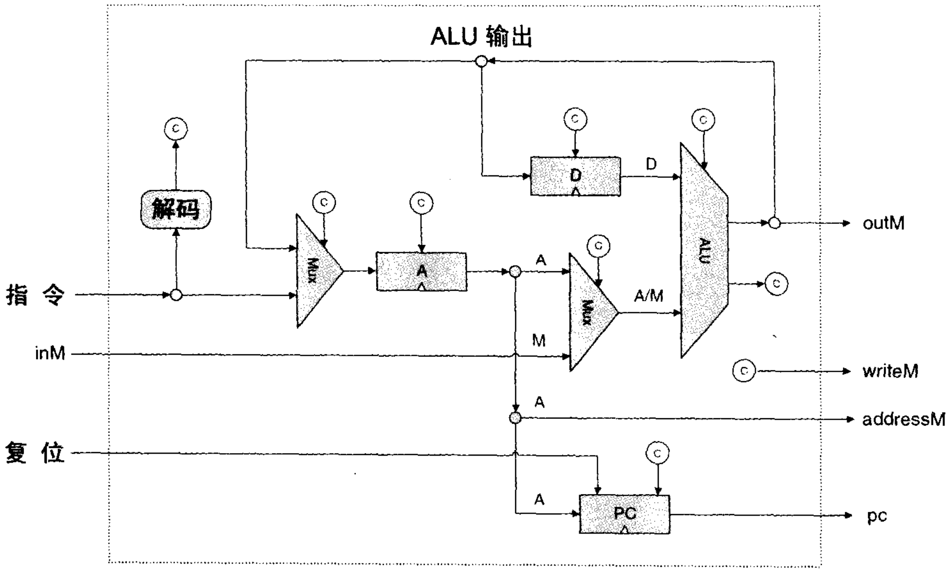


图 5.9 CPU 一种推荐的实现形式。该框图仅仅显示了数据和地址通道，也就是将数据和地址信息从一个地方传输到另一个地方的导线。除了给出一些输入/输出控制位（用带圆圈的“c”标示），该框图并没有显示 CPU 的控制逻辑。因此，它不能被看作为完整的芯片框图

CPU 包括执行 Hack 指令的 ALU、一组寄存器和一些用于取指令和对指令解码的控制逻辑。因为几乎所有这些硬件元素都已经在前面的章节中构建好了，所以这里的关键问题是如何正确地连接它们。图 5.9 给出了一种可行的方案。

图 5.9 中没有给出的部分是 CPU 的控制逻辑，用来执行下面的任务：

- **指令解码**（*Instruction decoding*）：解析出指令所代表意思（指令的功能）。
- **指令执行**（*Instruction execution*）：发信号指示计算机的各个部分应该做什么工作来执行指令（指令的功能）。
- **读取下一条指令**（*Next instruction fetching*）：指出下一步执行哪一条指令（指令的功能以及 ALU 的输出）。

（本书后续所提到的“CPU 实现”就是指图 5.9 所展示的架构。）

指令解码（Instruction Decoding） CPU 输入的 16 位指令可以代表一条 A-指令或者一条 C-指令。为了解析该 16-位字的内容，可以将其分解为“`i xx a cccccc ddd jjj`”。其中 `i`-位代表指令类型，0 代表 A-指令，1 代表 C-指令。如果是 C-指令，那么 `a`-位域和 `c`-位域共同表示 *comp* 部分，它指明要执行什么计算，`d`-位域表示 *dest* 部分，`j`-位域表示 *jump* 部分。如果是 A-指令，那么除了 `i`-位之外的 15 位就被解释成常数。

指令执行（Instruction Execution） 指令的各个域（`i`-位域、`a`-位域、`c`-位域、`d`-位域、`j`-位域）会被同时发送到 CPU 中的各个组件，这样各组件就会根据指令的内容各自进行工作，协同执行指令。这里所指的指令可以是 A-指令或 C-指令，它们由机器语言规范描述。其中 `a`-位域决定 ALU 是把 A 寄存器的输入当作操作数，还是把内存单元的输入当作操作数，`c`-位域决定 ALU 执行什么函数，`d`-位域决定 ALU 的计算结果将保存到哪里。

读取下一条指令（Next Instruction Fetching） 在执行当前指令的同时，CPU 还可以确定下一条指令的地址，通过程序计数器的输出端发送该地址。这个任务的“驱动器”就是**程序计数器**（它是 CPU 内部结构的一部分），它的输出直接和 CPU 的 `pc` 输出端口连接。这就是第 3 章中构建的 PC 芯片（如图 3.5 所示）。

大部分时候程序员都希望计算机去取程序中下一条指令并予以执行。因此，设 t 是当前的单位时间，默认的程序计数器操作应该是 $PC(t)=PC(t-1)+1$。当我们要执行 *goto n* 操作时，机器语言首先要将 A 寄存器置为 n（通过一条 A-指令实现），然后执行一条跳转指令（由后续的 C-指令的 `j`-位指定）。因此，我们的任务是提出一个能实现如下逻辑的硬件方案：

If jump(t) then $PC(t)=A(t-1)$
else $PC(t)=PC(t-1)+1$

为了方便起见（实际是经过精心设计的），该 jump 控制逻辑很容易通过我们给出的 CPU 来实现。前面提到的 PC 芯片接口（参见图 3.5）有一个 `load` 位来控制它对输入值的接收。因此，为了执行期望的 jump 控制逻辑，先要将 A 寄存器的输出端和 PC 的输入端相连接。余下的问题就是决定 PC 何时从 A 寄存器接收数据，也就是什么时候去执行 jump。这是一个包含两个信号的函数：（a）当前指令的 `j`-域，它用来指定在什么条件下才执行 jump；（b）ALU 的输出状态位，它用来指示 jump 条件是否满足。如果要执行一个 jump，就应该将 A 寄存器的输出值加载给 PC。否则，PC 应该加 1。

另外，如果希望计算机重新去执行程序，所要做的就是将程序计数器置为 0。这就是为什么在我们提出的 CPU 实现方案中，直接将 CPU 的 reset 输入和 PC 芯片的 reset 管脚相连接。

5.3.2　内存
Memory

根据前面的介绍，Hack 平台的 Memory 芯片主要由三个底层芯片构成：RAM16K、Screen、Keyboard。同时我们必须通过这 3 个底层芯片来实现一个统一的逻辑地址空间来满足程序员的要求，这个空间从地址 0 到 24576（`0x0000` 到 `0x6000`，如图 5.7 所示）。Memory 芯片的实现应该创建这样的连续空间。在第 3 章中介绍了将若干小容量 RAM 芯片组合连接为一个较大容量的 RAM 芯片（参看图 3.6，以及相应的 *n*-寄存器内存（*n-register memory*）的介绍），在这里可以采用相同的技术来实现这个连续空间。

5.3.3　计算机
Computer

一旦实现并测试了 CPU 和 Memory 芯片，整个计算机的构建就变得简单了。图 5.10 给出了一种可能的实现方案。

5.4　观点
Perspective

根据本书所一贯倡导的风格，Hack 计算机是最小的系统，或称最简陋的系统。典型的计算机平台有更多的寄存器、更多数据类型、更强大的 ALU、更丰富的指令集。然而，这些主要还是数量上的区别。定性来看，Hack 和大多数数字计算机一样，因为它们都依据相同的设计理念：冯·诺依曼体系结构。

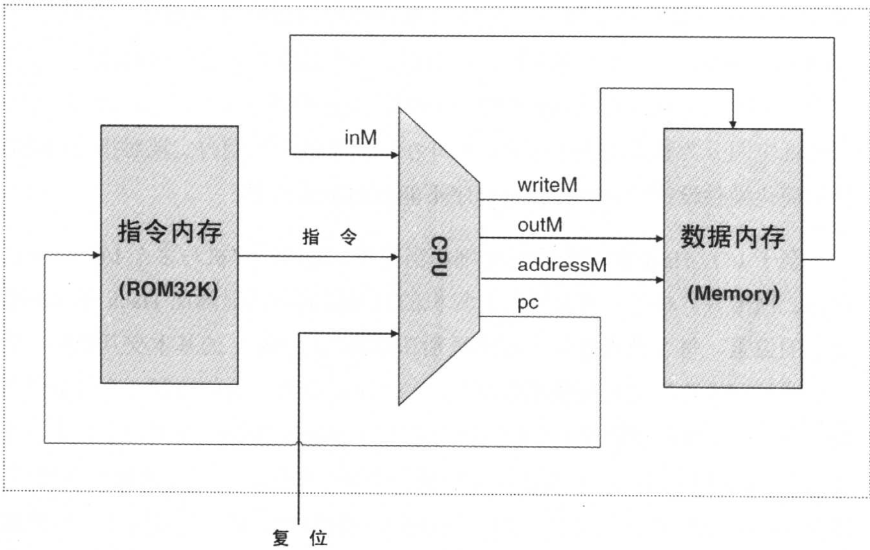


图 5.10 最高抽象级的 Computer 芯片实现形式

在功能方面，计算机系统可以被划分为两个类型：**通用计算机**（*general-purpose computers*），简单地用来执行一个个程序；**专用计算机**（*dedicated computers*），通常嵌入在其他系统中（比如手机、游戏机、数码相机、武器系统、制造设备等等）。对于一些特殊的应用，一个单一的程序会被烧写进专用计算机的 ROM 中，且该程序是唯一能被执行的程序（比如，在游戏机中，游戏软件存储在外部的游戏卡中，它实际上就是可更换的 ROM 模块，封装在漂亮的包装里）。除了这个显著的区别之外，通用计算机和专用计算机具有相同的结构：存储式程序、“获取-解码-执行”逻辑、CPU、寄存器、程序计数器等。

不同于 Hack，大多数通用计算机使用单一的地址空间用于存储数据和指令。在这样的结构中，指令中描述的指令地址和可选择的数据地址必须被发送到相同的目的地：共享地址空间的单一地址输入。显然这不能同时完成。标准的解决方法是将计算机的实现基于双周期控制逻辑。在**读取循环**（*fetch cycle*）中，从内存的输入端输入指令的地址，这样可

立即从内存中取得想要指令，并将该指令暂存到**指令寄存器**中。在接下来的**执行循环**（*execute cycle*）中，指令被解码。从中得出操作数所在内存单元的地址，并将该地址从内存的地址输入端输入，从而使指令可以操纵指定的内存单元。相比之下，Hack 体系结构将地址空间分为独立的两个部分，允许在同一个时钟周期内实现对指令的读取和执行。这样的简化硬件设计带来的代价是程序不能被动态地改变。

至于 I/O，Hack 的键盘和屏幕则相当简单。通用计算机与多个 I/O 设备（比如打印机、磁盘、网卡等）相连。并且通用计算机的显示器的功能显然比 Hack 的屏幕强大，它具有更多的像素，每个像素有若干亮度级别和颜色。跟 Hack 的基本交互原则一样，每个像素是通过驻留内存的二进制值来控制的：与 Hack 中单一的位控制一个像素的黑或白不同的是，在通用计算机中，若干个位用来控制三个基本色的亮度级别，由此来产生该像素的最终颜色。同样地，Hack 屏幕的内存映像是十分简单的，它直接将像素映像到内存的位中。然而大多数现代计算机允许 CPU 发送高级图像指令到用于控制显示器的**显卡**（*graphic card*）上。这样，CPU 再也不用为直接画一些类似于圆圈和多边形的图像而感到麻烦了，显卡使用其内嵌的芯片组来完成这个任务。

最后应该强调的是，设计计算机硬件时，大部分的努力和创造性都致力于使计算机获取更好的性能。因此，关于硬件体系结构的课程和教材都在围绕这些问题展开讨论，比如实现内存层级、更好地与 I/O 设备交互、流水线操作、并行化、指令预取以及本章中没有介绍的其他优化技术。

从发展的历程看，为了加强处理器性能而付出的努力引出了两个主要的硬件设计派系。**复杂指令集计算机**（CISC，*Complex Instruction Set Computing*）方法的倡导者认为，为了取得更好的性能，必须提供丰富和详细的指令集。与此相反的是，**精简指令集计算机**（RISC，*Reduced Instruction Set Computing*）阵营为了尽可能的提升硬件速度，使用了较简单的指令集。Hack 计算机并不加入这场争辩，既不具有强大的指令集也没有采用特殊的硬件加速技术。

5.5 项目
Project

目标 构建 Hack 计算机平台，最后得到最高级的 Computer 芯片。

资源 本项目中你所需要的工具仅仅是硬件仿真器和测试脚本。计算机平台应该用 HDL 语言来实现。

约定 本项目所构建的计算机平台应该能够执行用 Hack 机器语言编写的程序。为了验证这个能力，让你构建的 Computer 芯片运行这里给出三个程序。

组件测试 我们为 Memory 和 CPU 芯片分别提供了测试脚本和比较文件。在构建和测试整个 Computer 芯片之前，完成这些芯片的测试工作是很重要的。

测试程序 测试整个 Computer 芯片的一般方法是，让它去执行一些用 Hack 机器语言编写的范例程序。为了运行这样的测试，读者可以编写测试脚本，该脚本将 Computer 芯片加载到硬件仿真器，将程序从外部的文本文件导入到它的 ROM 芯片中，然后运行足够周期的时钟以执行程序。下面列出了运行三个这样的测试所需要的所有的文件：

1. `Add.hack`：将两个常数 2 和 3 相加，结果写入 RAM[0]。
2. `Max.hack`：计算 RAM[0]和 RAM[1]的最大值，结果写入 RAM[2]。
3. `Rect.hack`：在屏幕的左上角画一个宽为 16 个像素、长为 RAM[0]的矩形。

在用这些程序测试你的 Computer 芯片之前，请阅读与程序相关的测试脚本，理解输入到仿真器中的指令，可以参看附录 B。

步骤 按如下顺序来构建计算机：

- **内存**：由三个芯片组成：RAM16K、Screen 和 Keyboard。Screen 和 Keyboard 芯片在前面都已经构建好，可以直接使用。RAM16K 芯片虽然在第 3 章的项目中已经构建了，但是我们推荐使用它的内置版本，因为内置版本提供了友好的 GUI 用于调试。

- **CPU**：可以根据图 5.9 介绍的方案来构建，使用第 2 章构建的 ALU 和第 3 章构建的寄存器芯片。我们推荐使用这些芯片的内置版本，特别是 A 寄存器和 D 寄存器。这些芯片跟第 3 章中介绍的 Register 芯片功能一样，另外它们还支持 GUI 显示。
- **指令内存**：使用内置的 ROM32K 芯片。
- **计算机**：最高级的 Computer 芯片可以由前面提到的芯片组成，参考图 5.10 作为设计蓝本。

硬件仿真器 跟第 1 章到第 3 章中的项目一样，本项目的所有芯片（包括最高级的 Computer 芯片）都可以用与本书配套的硬件仿真器来实现并测试。图 5.11 是在 Computer 芯片上测试 `Rect.hack` 程序的屏幕截图。

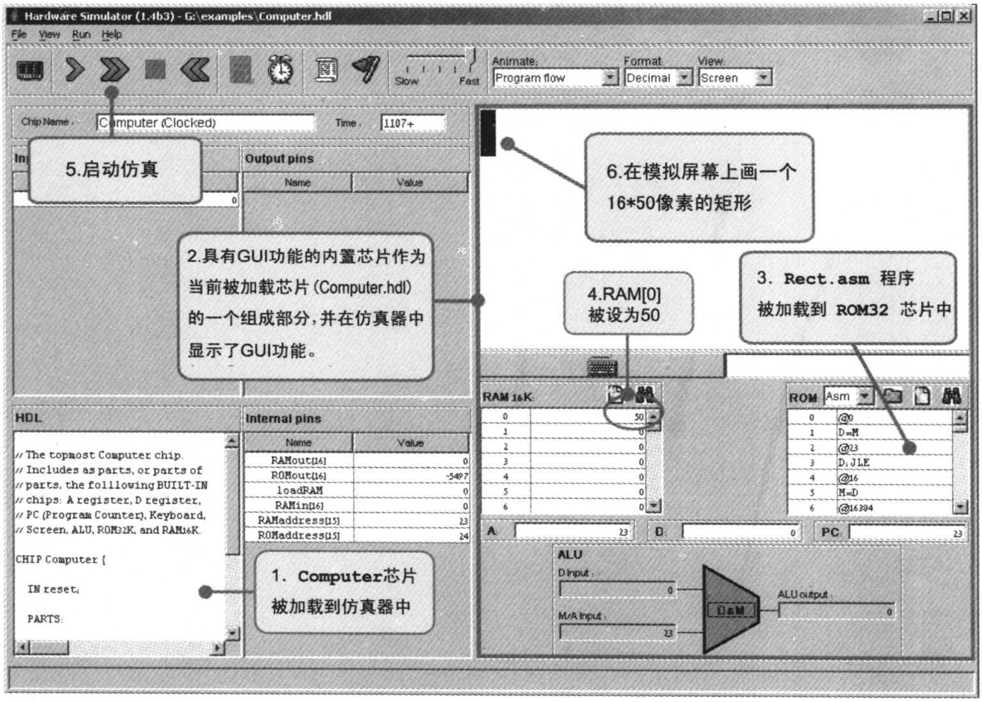


图 5.11　在硬件仿真器上测试 Computer 芯片。Rect 程序在屏幕的左上角上画了一个宽为 16 长 RAM[0]的矩形。注意，该 Rect 程序本身是正确的！因此，如果它没有正常工作，那就意味着运行该程序的计算机平台（Computer.hdl 和/或其他底层部分）存在问题

第 6 章　汇编编译器

Assembler

What's in a name? That which we call a rose by any other name would smell as sweet.
名称有什么关系呢？玫瑰不叫玫瑰，依然芳香如故。
——莎士比亚，罗密欧与朱丽叶

本书的前半部分（第 1 章到第 5 章）描述并构建了计算机的**硬件平台**（*hardware platform*）。后半部分（第 6 章到第 12 章）将集中讨论计算机的**软件阶层体系**（*software hierarchy*），直到最后为简单的面向对象编程语言开发编译器和基本的操作系统为止。在这个软件阶层体系中，最基本的模块就是**汇编编译器**（*assembler*）。第 4 章中介绍了机器语言的两种表现形式，即**汇编形式**和**二进制形式**。本章将介绍编译器如何系统地将汇编语言编写的程序翻译成二进制形式。随着本章内容的深入，我们还要阐述如何开发 **Hack 汇编器**——该程序产生的二进制代码能在第 5 章所建构的硬件平台上运行。

因为符号化汇编命令（symbolic assembly commands）与其对应的二进制代码之间的关系是很简单的，所以编写汇编编译器（使用某种高级语言编写）并非是很难的任务。唯一的复杂性在于：允许汇编程序使用符号来指代内存地址。我们希望编译器来管理这些用户自定义的符号（symbols），将它们解析成物理内存地址。一般可以使用**符号表**（*symbol table*）来完成这个任务，这种符号表是经典的数据结构，应用在很多软件编译过程中。

Hack 汇编编译器不是软件层级的终极目标，但它为任何汇编编译器的构建过程中关键的软件工程原则提供了简单明了的示范。而且，编写汇编编译器只是本书后半部分 7 个软件开发项目中的第一步。不同于用 HDL 实现的硬件项目，构建翻译程序（汇编编译器，

虚拟机和编译器）的软件项目是用某种编程语言来实现的。在每个项目中，我们提供中性语言 API 和一个详细的、按部就班的测试计划，包括相关的测试程序和测试脚本。每个项目都是单独的模块，都可以进行独立开发和测试。

6.1　背景知识 Background

机器语言一般分为两类：**符号型**（*symbolic*）和**二进制型**（*binary*）。二进制码（比如，11000010100000011000000000000111）代表一条实际的机器指令，它能被底层硬件所理解。指令最左边的 8 位代表操作码（比如 LOAD），接着的 8 位代表寄存器（比如 R3），剩下的 16 位表示地址（比如 7）。根据硬件的逻辑设计和相应的机器语言，整个 32-位指令可以让硬件去执行操作“将 Memory[7]的内容加载到寄存器 R3 中。” 现代计算机平台支持数十个（即使不是成百上千个）这样的基本操作。如此一来，机器语言会变得相当复杂，因为其涉及到很多操作码、不同的内存寻址模式和不同的指令格式。

解决此复杂性的方法之一是，使用约定的语法来表示机器指令，例如用 LOAD R3,7 而不是 11000010100000011000000000000111 表示机器语言。由于将符号表示翻译成二进制码是直截了当的，所以允许用符号表示法来编写底层程序，并用计算机程序将底层程序翻译成二进制码是很有意义的。符号化的语言称为**汇编**（*assembly*），翻译程序称为**汇编编译器**（*assembler*）。汇编编译器对每个汇编命令的所有部分进行解析，将每个部分翻译成它对应的二进制码，并将生成的二进制码汇编成真正能被硬件执行的二进制指令。

符号（Symbols）　二进制指令用二进制码表示。根据定义，它们使用实际的数字来指代内存地址。比如，假设某个程序用变量 *weight* 来表示不同事物的重量，假设该变量已经被映射到计算机内存地址为 7 的内存单元内。在二进制码层级上，操纵 *weight* 变量的指令必须使用明确的地址 7 来指代它。然而，一旦进入汇编级，我们就可以允许编写命令 LOAD R3,weight 来取代 LOAD R3,7 。两种情况都会执行相同的操作：“将 R3 设为 Memory[7] 的内容”。同样，假设在程序某一处标有符号 loop 来指代地址 250，那么我们

可以使用命令 goto loop 来取代 goto 250 ，假设在程序的某一处标有符号 loop 来指代地址 250 。符号在汇编程序通常有两个用途：

- **变量**（*Variables*）：程序员可以使用符号的变量名称，翻译器会"自动地"为其分配内存地址。需要注意的是，这些地址的实际值是没有意义的，只要在程序的整个编译过程中，每个符号始终被指代为同一地址就可以了。
- **标签**（*Labels*）：程序员可以在程序中用符号来标注不同的位置。比如，可以用标签 loop 来指代特定代码段的起始地址。程序中的其他命令就可以有条件或无条件地执行 goto loop 指令。

在汇编语言中引入符号（symbols）意味着，汇编编译器必须比简陋的文本处理程序更强大。好在将约定的符号翻译成约定的二进制码并不太复杂。同时，用户定义的变量名称和符号标签（symbolic labels）与实际内存地址的映射则不是那么简单。事实上，这个确定符号地址的任务是从硬件层级上升到软件层级过程中遇到的第一个挑战。下面的例子介绍了该任务和通常的解决方法。

符号解析（Symbol Resolution） 图 6.1 列出了一个用某种低级语言编写的简单程序。该程序包含 4 个用户自定义符号：2 个变量名称（i 和 sum）；2 个标签（loop 和 end）。那么，如何将其系统化地转换为不包含符号的代码呢？

我们首先来制定两个任意性规则：其一，翻译后的代码将被存储到计算机中起始地址为 0 的内存中；其二，变量将会被分配到起始地址为 1024 的内存中（这些规则依赖于特定的目标硬件平台）。接下来，我们构建一个**符号表**（*symbol table*）。在源代码中，每遇到一个新符号 *xxx* ，我们就在符号表中添加一行（*xxx*, *n*）。根据规则约定，*n* 是分配给对应符号的内存地址。在符号表建立完成之后，我们利用它来将程序翻译成无符号的版本。

根据我们定义的规则，变量 i 和 sum 被分配的内存地址分别是 1024 和 1025。当然，只要程序中对 i 的所有引用都指代其所对应的同一物理地址，并且对 sum 的所有引用也都指代同一物理地址，那么为这两个变量各自分配其他任意地址也是可以的。剩余的代码就不用解释了，只是最后一条语句需要解释一下：该指令将让计算机进入无限循环。

用符号描述的代码

```
       // 计算 sum=1+…+100
00         i=1
01         sum=0
       loop:
02         if i=101 goto end
03         sum=sum+i
04         i=i+1
05         goto loop
       end:
06         goto end
```

符号表

i	1024
sum	1025
loop	2
end	6

（假设从 Memory[1024] 内存单元开始分配变量）

经过符号解析后的代码

```
00  M[1024]=1  // (M=memory)
01  M[1025]=0
02  if M[1024]=101 goto 6
03  M[1025]=M[1025]+M[1024]
04  M[1024]=M[1024]+1
05  goto 2
06  goto 6
```

假设每个用符号描述的命令被翻译成内存中的一个 word）

图 6.1 使用符号表来解析符号。左边的行数不是程序的组成部分——它们仅仅表示程序中真实指令的行数（注释和标记不算指令）。一旦有了正确的符号表，那么符号的解析工作就很简单了

这里有三点是需要说明的。首先，要注意到我们定义的变量分配规则决定了能运行的程序最多只能有 1024 条指令。然而实际的程序（比如操作系统）显然要大很多，因此存储变量的基地址也应该设得距离代码存储区更远一些。其次，“每条源代码命令映射到一个字（word）”的假设恐过于天真。一般来说，某些汇编命令（比如，`if i=101 goto end`）会被翻译成好几条机器指令，因此它会占据好几个内存单元。为了解决此问题，翻译程序会记录每条源代码产生的字的个数，然后相应地更新它的“指令内存计数器（Instvachion memory counter）”。

最后，对于“每个变量用一个单独的内存单元来表示”的假设可能也不切实际。编程语言支持多种类型的变量，它们在目标计算机上占用不同的内存空间。比如，C 语言数据类型 `short` 和 `double` 分别代表 16-位和 64-位数字。当 C 程序在 16-位机器上运行时，`short` 变量将占用 1 个单独内存单元，`long` 变量将占用 4 个连续单元组成的块。因此，当为变量分配内存空间时，翻译程序必须考虑它们的数据类型和硬件内存单元的宽度。

汇编编译器（Assembler） 汇编程序在被计算机执行之前，必须被翻译成计算机的二进制机器语言。 翻译任务是由称为汇编编译器的程序来完成。 汇编编译器的输入是一串

汇编命令，然后产生一串等价的二进制指令作为输出。生成的代码被加载到计算机的内存中然后被硬件执行。

可见，汇编编译器实际上主要是个文本处理程序，设计目的是用来提供翻译服务。编写汇编编译器的程序员必须有完整的汇编语法说明文档和相应的二进制代码。有了这样的约定（通常称为机器语言规范），就不难编写程序，让其对每个符号命令执行下面的任务（顺序无关，不分先后）：

- 解析出符号命令内在的域。
- 对于每个域，产生机器语言中相应的位域。
- 用内存单元的数字地址来替换所有的符号引用。
- 将二进制码汇编成完整的机器指令。

其中三个任务（解析、代码生成和汇编）是相当容易实现的。而符号处理则相对较为复杂，是汇编编译器的主要功能。这个功能在前面的小节里面描述过。下面两个部分将会详细介绍 Hack 汇编语言及其汇编编译器的实现方式。

6.2 Hack 汇编到二进制的翻译规范详述 Hack Assembly-to-binary Translation Specification

Hack 汇编语言和它对应的二进制表示在第 4 章中已经介绍过了。为了便于参考，这里给出了该语言规范简洁、正式的版本。该规范可以被看作是 Hack 汇编编译器必须满足的规约。

6.2.1 语法规约和文件格式 Syntax Conventions and File Formats

文件名称 习惯上，二进制机器代码表示的程序和汇编代码表示的程序被存储在后缀名分别为“hack”和“asm”的文本文件中。因此，Prog.asm 文件会被汇编编译器翻译成 Prog.hack 文件。

二进制代码（`.hack`）**文件**　二进制代码文件由文本行组成。每一行由 16 个 0/1 组成的 ASCII 码构成一个序列，该序列对一个单一的 16-位机器语言指令进行编码。因此，文件中的所有行在整体上代表一个机器语言程序。当机器语言程序被加载进计算机的指令内存中时，文件的第 n 行二进制码被存储到地址为 n 的指令内存单元内（设程序命令行的计数和指令内存的起始地址都是从 0 开始）。

汇编语言（`.asm`）**文件**　汇编语言文件由文本行组成，每一行代表一条**指令**（*instruction*）或者一个**符号声明**（*symbol declaration*）。

- **指令**（*Instruction*）：A-指令或 C-指令，会在 6.2.2 节介绍。
- `(Symbol)`：该伪命令将 `Symbol` 绑定到该程序中下一条命令的地址上。因为它并不产生机器代码，所以称之为“**伪命令**（*pseudo-command*）”。

（下面的语法规定仅适合于汇编程序。）

常数（Constants）**和符号**（Symbols）　**常数**必须是非负的，用十进制表示。用户定义的**符号**可以是由字母、数字、下画线（_）、点（.）、美元符号（$）和冒号（:）组成的字符序列，但是不能以数字开头。

注释　以两条斜线（//）开头的文本行被认为是一条注释，注释不会被计算机执行。

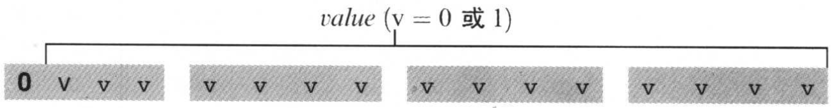
空格　空格字符和空行被忽略。

大小写习惯　所有的汇编助记符必须大写。剩余的部分（用于自定义标号和变量名称）是区分大小写的。一般的习惯是，标签（labels）大写，变量名称小写。

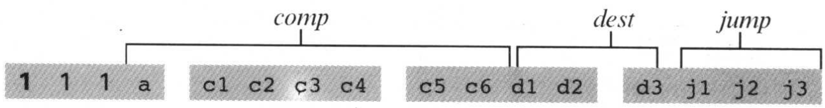
6.2.2 指令
Instructions

Hack 机器语言包含两种指令类型，分别称为**寻址指令**（A-指令，*Addressing Instruction*）和**计算指令**（C-指令，*Compute Instruction*）。指令格式如下：

A-指令：@*value*　　　　// *value* 是一个非负十进制数
　　　　　　　　　　　　// 或表示该数值的符号。

二进制：

C-指令：　*dest*=*comp*; *jump*　　//*dest* 或 *jump* 域都可以为空。
　　　　　　　　　　　　//如果 *dest* 为空，则 “=” 被省略；
　　　　　　　　　　　　//如果 *jump* 为空，则 “;” 被省略。

二进制：

要将三个域 *comp*，*dest*，*jump* 翻译成各自对应的二进编码，可以参看下面的三个表。

comp (when a=0)	c1	c2	c3	c4	c5	c6	***comp*** (when a=1)
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

dest	d1	d2	d3
null	0	0	0
M	0	0	1
D	0	1	0
MD	0	1	1
A	1	0	0
AM	1	0	1
AD	1	1	0
AMD	1	1	1

jump	j1	j2	j3
null	0	0	0
JGT	0	0	1
JEQ	0	1	0
JGE	0	1	1
JLT	1	0	0
JNE	1	0	1
JLE	1	1	0
JMP	1	1	1

6.2.3 符号

Symbols

Hack 汇编命令通过使用常数或符号来指代内存单元（地址）[1]。汇编程序中的符号来源于三个方面。

预定义符号（Predefined Symbols） 任何 Hack 程序允许使用下面的预定义符号。

Label	*RAM address*	*(hexa)*
SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0-R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

注意到表中的 RAM 地址 0 到 5 可以使用两个预定义符号来指代。比如说，R2 或者 ARG 都可以被用来指代 RAM[2] 。

标签符号（Label Symbols） 伪命令 (Xxx) 定义符号 Xxx 来指代存有程序中下一条命令在指令内存中的地置。每个标签只能被定义一次，可以在程序中的任何地方使用，甚至可以在它自身被定义的那一行之前使用。

[1] 从本节开始，若未经特殊说明，内存单元可以用名词“地址”代替，如变量 Val1 的地址是 0X5B46，这包含有两层意思：1）说明 Val1 的物理地址是数值 0X5B46；2）Val1 代表的变量存储在地址为 0X5B46 的内存单元内。更好的例子是汇编语言中的访问语句：MOV AX, Val1 等价于 MOV AX, (0X5B46)。——审校者

变量符号（Variable Symbols） 任何出现在汇编程序中的符号 Xxx，如果它不是预定义符号也没有在其他地方使用(Xxx)命令，那么它就被当作是变量。每当程序遇到一个新的变量时，就把它连续地映射到从 16 开始的内存中去。

6.2.4 范例

Example

第 4 章中介绍了从 1 加到 100 的整数累加程序。图 6.2 再次给出了这个例子，同时显示了其汇编版本和二进制版本。

汇编代码（**Prog.asm**）

```
// Adds 1 + ... + 100
      @i
      M=1    // i=1
      @sum
      M=0    // sum=0
(LOOP)
      @i
      D=M    // D=i
      @100
      D=D-A  // D=i-100
      @END
      D;JGT  // if (i-100)>0 goto END
      @i
      D=M    // D=i
      @sum
      M=D+M  // sum=sum+i
      @i
      M=M+1  // i=i+1
      @LOOP
      0;JMP  // goto LOOP
(END)
      @END
      0;JMP  // 无限循环
```

→ 汇编编译器 →

二进制码 （**Prog.hack**）

```
(本行应被删除)
0000 0000 0001 0000
1110 1111 1100 1000
0000 0000 0001 0001
1110 1010 1000 1000
(本行应被删除)
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0110 0100
1110 0100 1101 0000
0000 0000 0001 0010
1110 0011 0000 0001
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0001 0001
1111 0000 1000 1000
0000 0000 0001 0000
1111 1101 1100 1000
0000 0000 0000 0100
1110 1010 1000 0111
(本行应被删除)
0000 0000 0001 0010
1110 1010 1000 0111
```

图 6.2 同一个程序对应的汇编代码和二进制代码

6.3 实现 Implementation

Hack 汇编编译器的输入是名为 `Prog.asm` 的文本文件，该文件包含一个 Hack 汇编程序，编译器输出名为 `Prog.hack` 的文本文件，该文件包含编译后的 Hack 机器代码。输入文件的名字作为一个命令行参数提供给汇编编译器。

```
提示符> Assembler Prog.asm
```

将每个独立的汇编命令翻译成其等价的二进制指令是直接并一一对应的过程。每个命令被单独地翻译。特别是，汇编命令中的每个域会根据 6.2.2 小节中的参照表被翻译成对应的编码，命令中的每个符号也会根据 6.2.3 小节中的描述被解析成相应的数字地址。

我们这里提出一个基于 4 个模块的汇编编译器的实现：**语法分析器**（*Parser*）模块用来对输入文件进行语法分析；**编码**（*Code*）模块用来提供所有汇编命令所对应的二进制代码；**符号表**（*Symbol Table*）模块用来处理符号。另外还有一个主程序用来驱动整个编译过程。

关于 API 的说明 汇编编译器的开发是 5 个软件构建项目中的第一个，也是构建翻译器（**汇编编译器、虚拟机和编译器**）层级中的第一步。鉴于读者可能采用不同编程语言来开发这些项目，我们将这里介绍的实现原则建立在"与语言无关的 API"之基础上。典型的项目 API 描述了几个**模块**（*modules*），每个模块都包含一个或多个程序。在一些面向对象语言（比如 Java、C++和 C#）中，一个模块通常对应一个**类**（*class*），一个**程序**（*routine*）通常对应一个**方法**（*method*）。在过程化语言中，程序（routines）对应**函数**（*functions*）、**子函数**（*subroutines*）或者**过程**（*procedures*），模块则对应处理相关数据的程序集合。在某些语言（比如 Modula-2）中，模块可被显式地表示，而在其他一些语言中则被隐式地表示（比如 C 语言中的 *file*），在另一些语言（比如 Pascal）中，模块没有相应的语言结构，它仅仅代表在概念上对程序进行分组。

6.3.1 *Parser* 模块 The *Parser* Module

语法分析器（parser）的主要功能是，将汇编命令分解为其所表达的内在含意（域和符号）。它的 API 如下表示：

Parser：封装对输入代码的访问操作。功能包括：读取汇编语言命令并对其进行解析；提供"方便访问汇编命令成分（域和符号）"的方案；去掉所有的空格和注释。

程　序	参　数	返回值	功　能
构造函数/初始化函数	输入文件/输入流	—	打开输入文件/输入流，为语法解析作准备
hasMoreCommands	—	Boolean	输入当中还有更多命令吗
advance	—	—	从输入中读取下一条命令，将其当作"当前命令"。仅当 hasMoreCommands()为真时，才能调用本程序。最初始的时候，没有"当前命令"
commandType	—	A_COMMAND, C_COMMAND, L_COMMAND	返回当前命令的类型： ■ A_COMMAND 当@Xxx 中的 Xxx 是符号或十进制数字时 ■ C_COMMAND 用于 dest=comp;jump ■ L_COMMAND（实际上是伪命令） 当(Xxx)中的 Xxx 是符号时
symbol	—	字符串	返回形如@Xxx 或(Xxx)的当前命令的符号或十进制值。仅当 commandType()是 A_COMMAND 或 L_COMMAND 时才能调用
dest	—	字符串	返回当前 C-指令的 dest 助记符（具有 8 种可能的形式） 仅当 commandType()是 C_COMMAND 时才能调用

续表

程　序	参　数	返回值	功　能
comp	—	字符串	返回当前 C-指令的 comp 助记符（具有 28 种可能的形式）。仅当 commandType() 是 C_COMMAND 时才能调用
jump	—	字符串	返回当前 C-指令的 jump 助记符（具有 8 种可能的形式）。仅当 commandType() 是 C_COMMAND 时才能调用

6.3.2 *Code* 模块
The *Code* Module

Code：将 Hack 汇编语言助记符翻译成二进制码。

程　序	参　数	返回值	功　能
dest	助记符（字符串）	3 bits	返回 dest 助记符的二进制码
comp	助记符（字符串）	7 bits	返回 comp 助记符的二进制码
jump	助记符（字符串）	3 bits	返回 jump 助记符的二进制码

6.3.3 无符号程序的汇编编译器
Assembler for Programs with No Symbols

建议将编译器的构建分为两个阶段。在第一阶段，编写汇编编译器来翻译无符号汇编程序。这可以通过前面描述的 Parser 和 Code 模块来实现。在第二阶段，将其扩展成具有符号处理能力的汇编编译器，下一节中会介绍。

在第一阶段的约定是：输入的 `Prog.asm` 程序不包含符号。这意味着：（a）在所有的地址命令`@Xxx`中，`Xxx` 常数是十进制数而不是符号；（b）输入文件不包含标记命令，也就是没有`(Xxx)`命令。

无符号汇编编译器程序的实现过程如下：首先，程序打开名为 `Prog.hack` 的输出文件。接下来，程序开始处理 `Prog.asm` 文件的每一行（汇编指令）。对于每条 C-指令，程序将翻译后的指令域的二进制码连接到一个单一的 16-位字上。然后，程序将这个字写入 `Prog.hack` 文件。对于每条 A-指令`@Xxx` ，程序将语法分析器返回的十进制常数翻译成对应的二进制表示，然后将得到 16-位字并写入 `Prog.hack` 文件。

6.3.4 *SymbolTable* 模块
The *SymbolTable* Module

因为 Hack 指令可能包含符号，作为翻译过程的一部分，必须为这些符号确定实际的地址。汇编编译器使用**符号表**（*symbol table*）来完成这个任务，符号表用来建立和维持符号与其地址之间的关联。**哈希表**（*hash table*）就是表示这种关系的经典数据结构之一。在大多数编程语言中，这样的数据结构都是作为标准库的一部分而存在的，因此没有必要从头来开发它。下面给出了它的 API。

SymbolTable：在符号标签（symbolic labels）和数字地址之间建立关联。

程 序	参 数	返回值	功 能
`Constructor`	—	—	创建空的符号表
`addEntry`	`symbol`（字符串），`address`（int）	—	将（symbol,address）配对加入符号表
`contains`	`symbol`（字符串）	Boolean	符号表是否包含了指定的 symbol ?
`GetAddress`	`symbol`（字符串）	int	返回与 symbol 关联的地址

6.3.5 有符号程序的汇编编译器
Assembler for Programs with Symbols

汇编程序允许在符号被定义之前使用符号标签（即 *goto* 命令的目的地）。此功能为汇编程序员带来了极大的便利，却使得汇编编译器开发者的工作变得更复杂。解决这个问题

的方法之一是，编写“**两遍**（two-pass）”汇编编译器，从头至尾地读取两次代码。第一遍读取时，汇编编译器构建符号表但并不产生代码。第二遍读取时，程序中遇到的所有标签符号所对应的内存地址都已经被记录在符号表中了，这个符号表就是在编译器在第一次读取程序的过程中建立的。因此，汇编编译器能用每个符号相关的含义（数字地址）来替换该符号，并产生最后的二进制码。

前面介绍过在 Hack 语言中有三种类型的符号：**预定义符号**（*predefined symbols*），**标签**（*labels*）和**变量**（*variables*）。符号表应该包含并处理所有这些符号。

初始化 根据 6.2.3 小节，用所有预定义符号和它们预分配的 RAM 地址对符号表进行初始化。

第一遍读取阶段 该阶段主要是在符号表中建立每条命令及其对应的地址。逐行处理整个汇编程序，构建符号表而不生成任何代码。处理程序的每一行时，利用数字来记录 ROM 地址——当前命令最终将被加载到这个地址中。这个数字从 0 开始，不管碰到 C-指令还是 A-指令都自动加 1，但是当遇到标签伪指令或注释时不发生变化。每次遇到一条伪指令（Xxx）时，在符号表上加一个新条目来将 Xxx 与最终用于存储程序中下一条指令的 ROM 地址关联起来。在这个阶段程序中所有标记和它们的 ROM 地址被加入到符号表中。程序的变量放在第二阶段处理。

第二遍读取阶段 现在重新对整个程序进行处理，对每一行进行语法分析。每次遇到符号化 A-指令时，即@Xxx 指令中 Xxx 是符号而不是数字时，就在符号表中查找 Xxx 。如果在符号表中找到了该符号，就用其对应的地址来替换该符号以完成命令的翻译。如果在符号表中没有找到该符号，那么它必定代表变量。为了处理这个变量，就在符号表中添加（Xxx，*n*），这里 *n* 代表下一个可使用的 RAM 地址。分配的 RAM 地址是连续数字，从地址 16 开始（紧接着预定义符号的地址之后）。

这样，汇编编译器的实现过程就完成了。

6.4 观点 Perspective

跟大多数汇编编译器一样，Hack 汇编编译器是相对简单的程序，主要用于文本处理。显然，较丰富的机器语言的汇编编译器相对复杂。一些汇编编译器会具有 Hack 所缺乏的更好的符号处理能力。比如，这些汇编编译器可以允许程序员将符号与特定的数据地址联系起来对符号执行“常数运算”（比如，使用 `table+5` 来指代从 `table` 所代表的起始地址的第 5 个内存位置），等等。此外，很多汇编编译器能够处理**宏命令**（*macro commands*）。宏命令是简单的机器指令序列。比如，Hack 汇编编译器可以扩展到能将约定的宏命令，`D=M[`*xxx*`]`，翻译成两个指令：`@`*xxx* 和 `D=M` 。显然，这样的宏命令能够在相当程度上，以较低的翻译代价来简化涉及常用操作的程序编写。

需要注意的是，单独的（stand-alone）汇编编译器很少在实际中用到。首先，人们很少编写汇编程序，这主要是由编译器来负责。编译器是自动机，它不必为生成符号的命令而费心，因为它可以直接产生二进制机器代码。另一方面，很多高级语言编译器允许程序员在高级程序中嵌入汇编语言代码部分。这个功能在 C 语言编译器中相当普通，它让程序员能直接控制底层硬件以达到优化的目的。

6.5 项目 Project

目标 开发汇编编译器，将用 Hack 汇编语言编写的程序翻译成 Hack 硬件平台能够理解的二进制代码。关于翻译过程的描述在 6.2 节中介绍过，汇编编译器必须实现这个过程。

资源 完成这个项目所需的唯一工具就是用来实现汇编编译器的编程语言。下面的两个工具可能会有用：与本书配套的汇编编译器和 CPU 仿真器。这些工具能够让你在开始构建汇编编译器之前，用现成的汇编编译器来进行实验。此外，本书配套的汇编编译器支

持逐行翻译的 GUI，允许与你的汇编编译器生成的输出进行代码比较。关于这个功能的更多信息请参看汇编编译器手册（与本书配套的软件包中的一部分）。

约定 当 `Prog.asm` 文件被加载进你的汇编编译器中时，该文件包含的有效的 Hack 汇编语言程序应该被翻译成正确的 Hack 二进制代码，并存储到 `Prog.hack` 文件中。你的汇编编译器产生的输出必须跟本书提供的汇编编译器产生的输出是相同的。

构建计划 建议分两个阶段来构建汇编编译器。首先，编写无符号汇编编译器，也就是仅能翻译无符号程序的汇编编译器。然后，扩展其功能使其具有符号处理能力。本书提供的测试程序有无符号和有符号两个版本，帮助测试你的汇编编译器。

测试程序 除了第一个程序之外，每个测试程序都有两个版本：`ProgL.asm` 是无符号版本，`Prog.asm` 是有符号版本。

Add：将常数 2 和 3 相加，结果保存在 R0 中。

Max：计算 max(R0，R1)，结果保存在 R2 中。

Rect：自屏幕的左上角画一个矩形。该矩形宽度为 16 个像素，高度为 R0 个像素。

Pong：单人乒乓球游戏。一个球在屏幕的“墙壁”上连续地反弹。游戏者通过敲击键盘上左右键控制球拍来击球。对于每次成功的击球，游戏者都会得一分，同时球拍也会缩小一点，这样使得击球的难度逐渐变大。如果游戏者没有击中球，游戏结束。按 Esc 键退出游戏。

Pong 程序是用 *Jack* 编程语言（第 9 章介绍 Jack 语言）编写的，然后被 **Jack 编译器**（第 10 至 11 章）翻译成汇编程序。虽然原始的 Jack 程序仅仅只有 300 行代码，然而可执行的 Pong 应用程序大概有 20 000 行二进制码，其中大部分代码是 Jack 操作系统（第 12 章）。在 CPU 仿真器中运行这个交互式程序是一件很慢的事情，所以不便于开发功能更强大的 Pong 游戏。在这里慢其实还有好处，因为这样你就可以从容地靠眼睛来捕捉该程序的图像行为。在本书后面的项目中，该游戏会运行得更快。

步骤 分两个阶段去编写和测试你的汇编编译器程序。可以利用与本书配套的汇编编译器来比较你自己的编译器输出。测试程序接下来会介绍。关于与本书配套的汇编编译器的更多信息，请参阅汇编编译器手册。

与本书配套的汇编编译器

图 6.3 展示了使用汇编编译器（能产生正确的二进制代码）来测试另一个汇编编译器（不一定能产生正确的二进制代码）的情况。假设 `Prog.asm` 是用 Hack 汇编语言编写的程序。首先，用提供的汇编编译器将其翻译为 `Prog.hack` 文件。然后，用另一个汇编编译器（你编写的汇编编译器）来将 `Prog.asm` 进行翻译，假设生成 `Prog1.hack` 文件。若

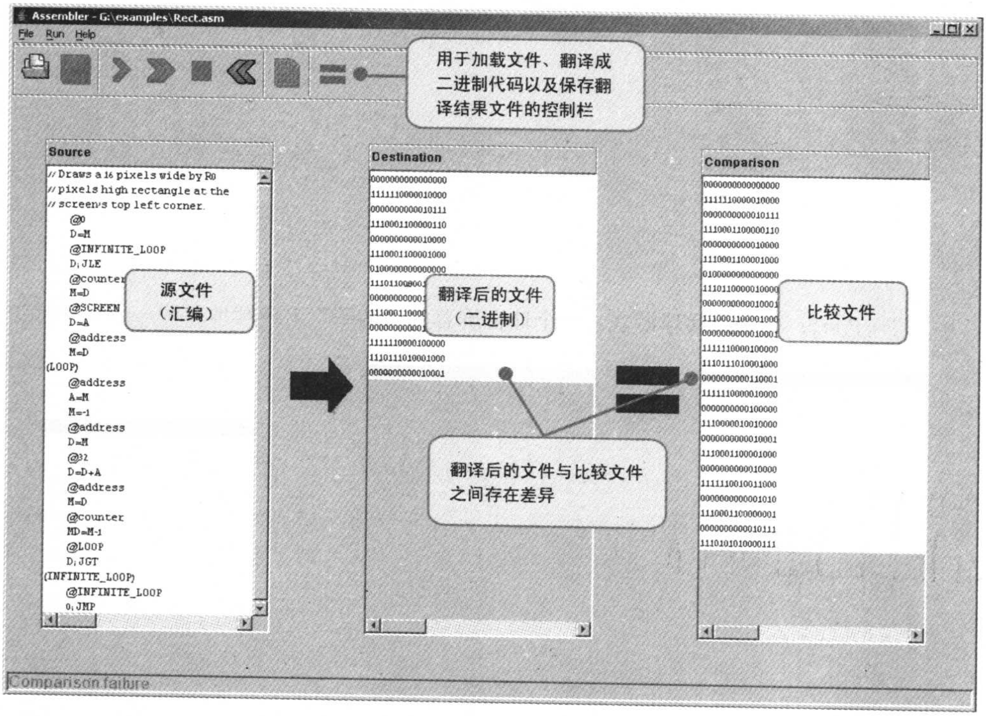


图 6.3 利用本书提供的编译器测试由其他汇编编译器生成的代码

第二个汇编编译器工作正确无误，那么 Prog.hack 与 Prog1.hack 的内容就应该相同。因此，测试新编写的汇编编译器的方法是：先将 Prog.asm 加载至已经提供的汇编编译器当中，再将 Prog1.hack 作为比较文件加载进去，然后翻译并比较两个生成的二进制文件（如图 6.3 所示）。如果对比发现有不同，那么生成 Prog1.hack 的那个新编写的汇编编译器就一定有问题，需要调试；如果对比未发现不同，那么新编写的汇编编译器就是正确无误的。

第 7 章　虚拟机 I：堆栈运算

Virtual Machine I: Stack Arithmetic

Programmers are creators of universes for which they alone are responsible. Universes of virtually unlimited complexity can be created in the form of computer programs.

程序员对其所创造的宇宙全权负责，因他们系这些宇宙的创造者。以计算机程序的形式，可创造出几乎无限复杂的宇宙。

——Joseph Weizenbaum，

Computer Power and Human Reason（《计算机能力与人类理性》，1974）

这一章介绍构建典型的基于对象（object-based）的高级语言**编译器**的最初步骤。我们分两个部分来介绍，每个部分涵盖两个章节。首先，高级语言将被翻译成中间代码（第 10、11 章），然后中间代码被转化为机器语言（第 7、8 章）。这种两层（two-tier）转换模型的思想比较古老，可以追溯到 20 世纪 70 年代。最近，由于被一些现代高级语言（比如 Java 和 C#）所采用，因此该模型得到了复兴。

基本思想如下所述：中间代码运行在**虚拟机**（*Virtual Machine*）上，而不是真实的硬件平台上。VM 是并不真实存在的抽象计算机，但是却能在其他的计算机平台上得以实现。为什么这种思想极具意义，有很多原因，其中一个原因在于代码的可移植性。因为 VM 能够在多目标平台上相对轻松地实现，因此基于 VM 的软件不经过修改源代码就可以在不同的处理器和操作系统上运行。VM 的实现可以通过多种途径：通过软件翻译器，通过特殊用途的硬件，或者通过把 VM 程序翻译成目标平台的机器语言。

本章介绍典型的 VM 结构，它是以 **Java 虚拟机**（*Java Virtual Machine*，JVM）的模型为蓝本的。通常，我们将注意力集中在两个方面：首先提出 VM 抽象，然后在 Hack 平台上来实现它。我们必须编写名为 **VM 翻译器**（*VM Translator*）的程序，用来将 VM 代码翻译成 Hack 汇编代码。与本书配套的软件包提供了另一种实现 VM 的方法，即称之为 **VM 仿真器**的程序，该程序应用 Java 语言在标准个人计算机上模拟 VM。

虚拟机模型一般配有一种**语言**（*language*），可以利用这种语言来编写 **VM 程序**。这里要介绍的 VM 语言包括 4 种类型的命令：算术命令，内存访问命令，程序流程控制命令和子程序调用命令。我们把这种语言的实现分成两个部分，每一部分都会在独立的章节和项目里面进行介绍。本章构建基本的 VM 翻译器，它能将 VM 的算术命令和内存访问命令翻译成机器语言。在下一章将对这个基本的 VM 翻译器进行扩展，加入程序流程和子程序调用功能。由此构建的完整的虚拟机，将作为第 10 至 11 章中所要构建的编译器的后端程序（backend）。

虚拟机的出现论证了计算机科学领域里面的很多重要思想。首先，将一台计算机在另一台机器上面进行仿真的概念，是这个领域里的基本思想之一，可以追溯到上世纪 30 年代的阿兰·图灵（Alan Turing）。其后的几十年里此概念得到了很多实际的应用，比如说，在计算机平台上应用模拟器仿真先前的计算机来实现代码的向上兼容。近来，虚拟机模型成为两种具竞争关系的主流架构的中心焦点——Java 体系和.Net 架构。这些软件环境相当复杂，要想了解其内部结构，方法之一是构建其 VM 内核的简单版本，正如我们这里做的一样。

这一章里面另一个重要主题是**堆栈处理**（*stack processing*）。**堆栈**（*stack*）是基本且精良的数据结构，存在于很多计算机系统和算法中。由于本章介绍的 VM 是基于堆栈的（stack-based），因此该 VM 也正为堆栈这个功能强大的数据结构提供了鲜活的实例。

7.1 背景知识 Background

7.1.1 虚拟机范型 The Virtual Machine Paradigm

高级语言程序能够在目标计算机上运行之前，它必须被翻译成计算机的机器语言。这个翻译工作——也就是**编译**（*compilation*）——是相当复杂的过程。通常，必须对任意给定的高级程序和其对应的机器语言编写专用的编译器。每种编译器编译的高级语言与编译之后的机器语言之间存在很强的依赖性。减少这种依赖性的方法之一是，将整个编译过程划分成两个几乎独立的阶段。在第一个阶段，高级程序被解析出来，其命令被翻译成一种

中间处理结果——既不是“高级”也不是“低级”的中间结果。在第二个阶段，这些中间结果被进一步翻译成目标硬件的机器语言。

从软件工程的角度来分解是非常吸引人的：第一阶段仅依赖于源高级语言的细节，第二阶段仅依赖于目标机器语言的细节。当然，两个编译阶段之间的接口（接口就是中间处理步骤的精确定义）必须仔细地进行设计。实际上，该接口的重要性之高，甚至应该将其单独定义为一种抽象计算机的语言。其实我们可以明确地描述这种**虚拟机**（*virtual machine*），其指令就是由高级命令分解而成的中间处理步骤。原来作为一个独立程序的编译器现在被分成两个独立的程序。第一个程序，仍然称为**编译器**（*compiler*），将高级代码翻译成中间 VM 指令，第二个程序将这个 VM 代码翻译成目标计算机硬件平台（简称“硬件平台”）的机器语言。

这个两阶段的编译模型已经通过这种或那种方式应用在很多编译器构建项目中。一些开发人员还定义了一门独立虚拟机语言，最著名的是在上世纪 70 年代由一些 Pascal 编译器生成的 *p*-**码**（*p-code*）。Java 编译器也是两层（two-tiered）结构，它生成运行在 JVM 虚拟机上（也称为 **Java 运行时环境**，*Java Runtime Environment*）的 *byte-code* 语言。最近，这种方法也被.NET 架构所采用。.NET 需要编译器来生成用 **IL 中间语言**（*Intermediate Language*）编写的代码，此中间语言运行在称为**通用语言运行时**（*Common Language Runtime*，CLR）的虚拟机上。

明确且正式的虚拟机语言概念有很多务实的优点。首先，仅需要替换虚拟机实现部分（有时候称为编译器的**后端程序**，*backend*）就能相对容易地得到不同硬件平台的编译器。因此，虚拟机在不同硬件平台之间的可移植性可以实现代码效率、硬件成本和程序开发难度之间的权衡。其次，很多语言的编译器能够共享 VM 后端程序，允许代码共享和语言互用性。比如，某种高级语言善于科学计算，而另一种在处理用户接口方面很突出。如果把两种语言编译到通用的 VM 层，那么通过使用约定的调用语法，其中一种语言的程序就能够很容易地调用另一种语言的程序。

虚拟机方法的另一个优点是模块化。VM 效率的每一个改善都会立即被所有构建于其上的编译器继承。同样地，每个安装有 VM 实现的数字设备都能够受益于现有软件的庞大基础，如图 7.1 所示。

7.1.2 堆栈机模型
The Stack Machine Model

像很多程序语言一样，VM 语言包含算术操作、内存访问操作、程序流程控制操作和子程序调用操作。有很多软件实体可作为 VM 语言的实现，但在选择中要考虑的关键是：**在 VM 操作中的操作数和结果应该驻留在哪里**。也许最“最干净利落”的解决方法是将其放在**堆栈**（*stack*）数据结构里面。

在**堆栈机**（*stack machine*）模型里，算术命令将其操作数从堆栈顶弹出，并将结果从栈顶压入。其他的命令将数据项从堆栈顶弹出，并转移到指定的内存单元，或反向操作之。经证明，这些简单的堆栈操作可以被用来计算任何数学或逻辑表达式。此外，任何程序，不管它用哪种程序语言编写，都能被翻译成等价的堆栈机语言。这样的堆栈机模型能被应用在 **Java 虚拟机**和接下来要介绍和构建的 VM 上。

基本堆栈操作 堆栈是抽象的数据结构，它支持两个基本操作：**压入**（*push*）和**弹出**（*pop*）。压入操作是从堆栈顶部压入一个元素，当有新的元素从栈顶压入时，原栈顶的元素就向下移动一个单位；弹出操作是将栈顶的元素弹出，原来位于被弹出元素之下的元素就向上移动一个单位到达栈顶。堆栈执行的是“**后进先出**”（*last-in-first-out*，LIFO）的存储模式，如图 7.2 所示。

可以看到，堆栈存取在很多方面不同于普通的内存访问。首先，堆栈只有顶部一个出入口，一次只能存/取一个元素。其次，读取堆栈是一种“丢失”操作：读取栈顶元素值的唯一方法是将其从堆栈中**移除**。相比之下，从普通内存单元读取值的行为就不会对内存的状态产生影响。最后，给堆栈添加新元素的操作就是直接将其压入栈顶，而不会改变堆栈的其余部分。然而，将值赋予一个普通的内存位置则是“丢失”操作，因为它覆盖了该内存单元内原来的值。

堆栈数据结构可以通过很多方法来实现。最简单的方法是创建一个数组（就是所谓的堆栈）和一个**堆栈指针**变量（称为 *sp*，*stack pointer*）指向位于栈顶的元素。

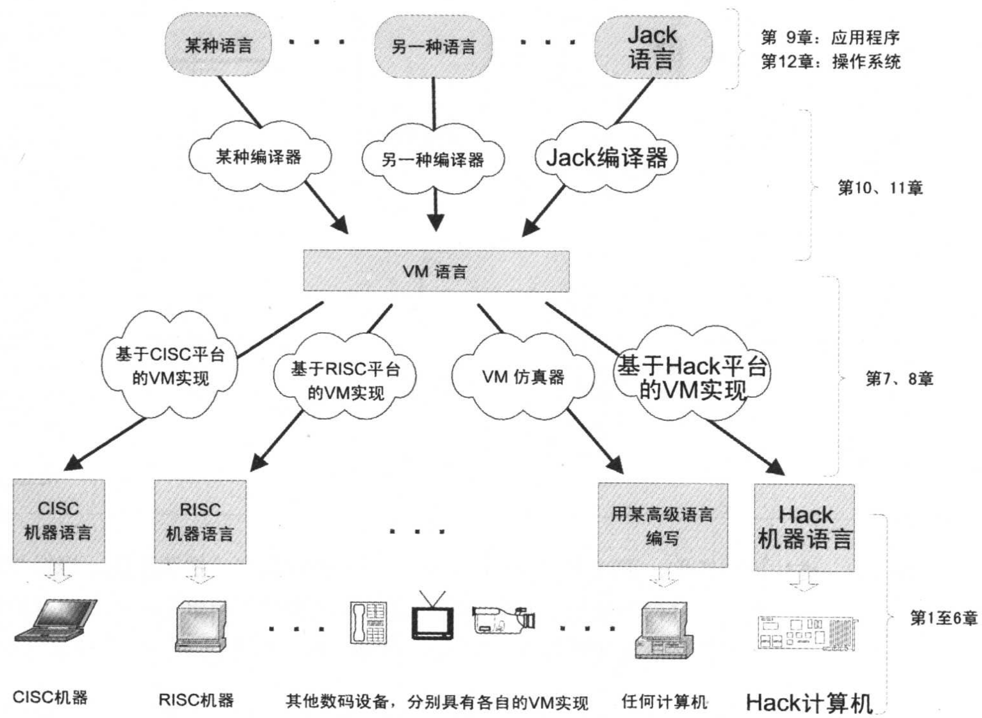


图 7.1 虚拟机应用实例。高级语言编写的程序一旦被编译成 VM 代码，那么就可以在任何配备了对应 VM 实现机制的硬件平台上运行。本章我们开始构建在 Hack 平台上的 VM 实现机制，并使用本图右边所描述的 VM 仿真器

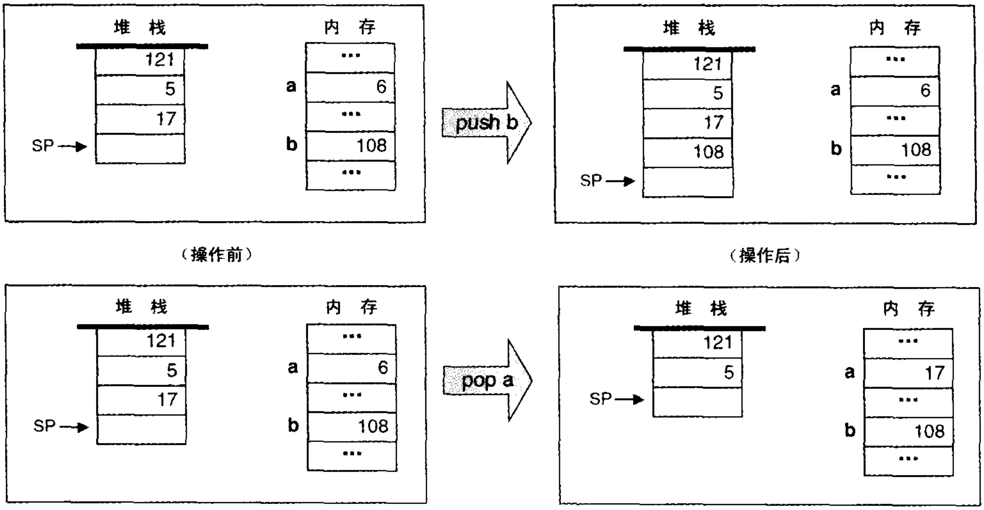


图 7.2　堆栈处理范例，描述了 *push* 和 *pop* 这两个基本操作。按照惯例，堆栈被描述为栈顶朝下，并向下增长的数据结构。紧挨着栈顶的内存单元由一个特殊的指针来引用，该指针称为堆栈指针，即 *sp*。符号 *a* 和 *b* 代表任意两个内存单元的地址

push x 命令将 *x* 存入数组中 *sp* 所指向的单元，然后将 *sp* 增加一个单位（比如，`stack[sp]=x; sp=sp+1`）。*pop* 命令首先将 *sp* 减少一个单位，然后返回存储在栈顶的值（比如，`sp=sp-1; return stack[sp]`）。

在计算机科学领域里，表达式简单优雅意味着强大的表达能力。结构简洁的堆栈模型是具备多功能的数据结构，它在许多计算机系统和算法中得到应用。在我们这里构建的虚拟机体系里面，堆栈主要有两个用途：首先，它被用来处理所有 VM 的算术和逻辑操作；其次，它使得子程序调用和相关的内存分配变得容易（下一章会讲到）。

堆栈运算　基于堆栈的运算是很简单的：操作数从堆栈弹出，对它们执行相关的操作，然后将结果压入堆栈。比如，这里给出了加法的处理过程：



其他操作（比如减、乘等）的堆栈运算方式是完全一样的。比如，高级语言程序的表达式 `d=(2-x)*(y-5)`，其基于堆栈的运算过程如图 7.3 所示。

布尔表达式的基于堆栈的运算方式也是一样的。比如，高级语言命令 `if(x<7) or (y=8) then...`。其基于堆栈的算术过程如图 7.4 所示。

前面所给出的例子论述了一个通用观点：任何算术表达式和布尔表达式（不管多复杂）都能被系统化地转化成一系列在堆栈上的简单操作，并被系统化地计算出来。因而，可以编写一个**编译器**，将这些高级算术和布尔表达式翻译成堆栈命令序列（正如我们将在第 10 至 11 章里面介绍的）。我们现在来讨论这些命令（7.2 节），并且介绍它们如何在 Hack 平台上实现（7.3 节）。

7.2 VM 规范详述，第 I 部分
VM Specification, Part I

7.2.1 概论
General

虚拟机是**基于堆栈**的（*stack-based*）：所有的操作都在堆栈上完成。它也是**基于函数**的（*function-based*）：一个完整的、应用 VM 语言编写的 VM 程序由若干个称为**函数**（*functions*）的程序单元组成，这些函数使用 VM 语言编写。每个函数都有自己独立的代码，并被独立地处理。VM 语言使用单一的 16-位数据类型，它能够表示整数、布尔类型，或者指针。该语言包含四种类型的命令：

- **算术命令** 在堆栈上执行算术和逻辑操作。
- **存储器存取命令** 在堆栈和虚拟内存单元之间转移数据。
- **程序流程命令** 使条件分支操作和无条件分支操作变得容易。
- **函数调用命令** 调用函数并返回调用处（即函数调用指令的下一条指令地址）。

```
// d=(2-x)*(y+5)
push 2
push x
sub
push y
push 5
add
mult
pop d
```

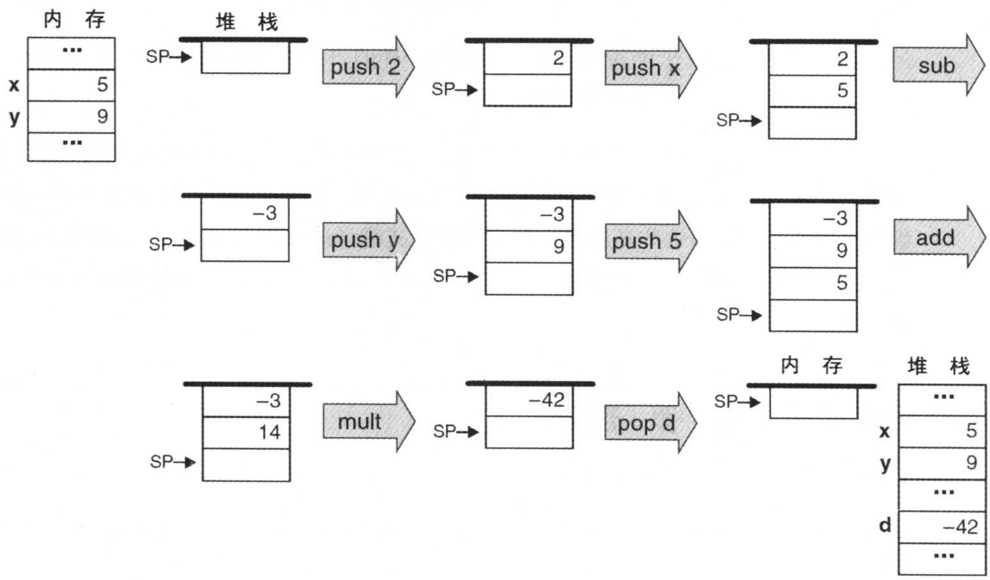


图 7.3 应用堆栈操作对算术表达式的计算过程。本例计算表达式 d=(2−x)*(y−5)的值，假设初始值 x=5，y=9

```
// if (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
```

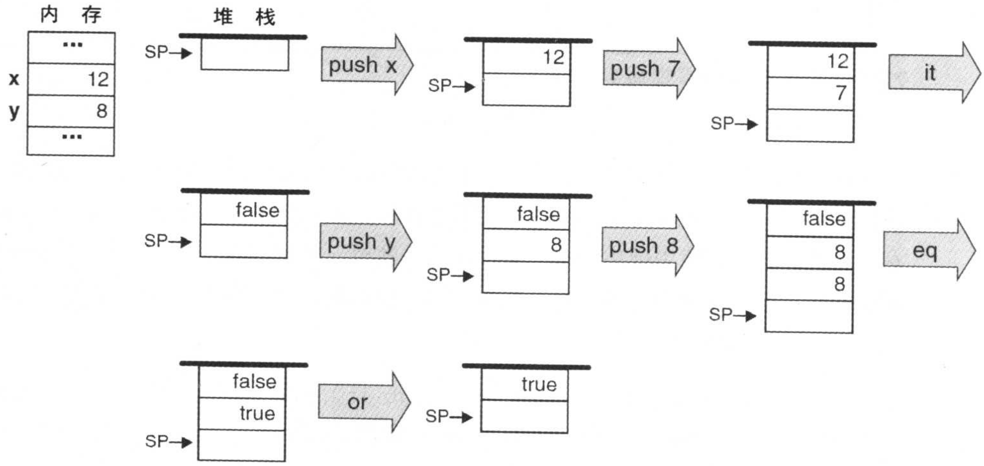


图 7.4 应用堆栈操作对逻辑表达式的计算过程。本例计算布尔表达式(x<7) or (y=8)的值，假设初始值 x=12，y=8

构建虚拟机是一件复杂的事情，所以我们把它分成了两个阶段。在本章里我们详细介绍**算术**命令和**内存访问**命令，构建仅实现这两种命令的基本 VM 翻译器。下一章将会详细介绍程序流程控制和函数调用命令，然后将我们构建好的翻译器扩展成一个包含所有这四种命令的虚拟机。

程序和命令结构 VM **程序**是由一个或多个扩展名为.vm 的**文件**构成的集合，而每个程序又包含一个或多个**函数**。从编译的角度来看，这些结构分别与面向对象编程语言中的**程序**（*program*）、**类**（*class*）和**方法**（*method*）的概念相对应。

在.vm 文件中，每一行一个 VM 命令，按照下面所列的格式之一来书写：

命令　　　　　　　　　（比如，add）
命令　参数　　　　　　（比如，goto loop）
命令　参数 1　参数 2　（比如，push local 3）

参数与参数之间，参数与命令之间必须用空格分离，空格的个数可以是任意的。每一行尾部可以添加以“//”开头的注释，它不会被程序执行。程序中允许出现空行，它同样也不会被执行。

7.2.2　算术命令和逻辑命令

Arithmetic and Logical Commands

VM 语言有 9 个面向堆栈的算术命令和逻辑命令。其中的 7 个命令是二元的：它们从堆栈中弹出两个元素，在其上执行二元函数操作，然后将结果压回堆栈。剩下的两个命令是一元的，只从堆栈中弹出一个元素，在其上执行一元函数操作，然后将结果压回堆栈。可以看到每个命令仅会用操作结果来取代它的操作数，而对堆栈的其他部分没有影响。图 7.5 给出了详细的细节。

图 7.5 列出的命令中有 3 个命令（eq,gt,lt）返回布尔类型的值。VM 分别用−1（0xFFFF）和 0（0x0000）来代表**真**（*true*）和**假**（*false*）。

7.2.3　内存访问命令

Memory Access Commands

命令	返回值（弹出操作数之后）	说　明	
add	$x + y$	整数加法	2-补码
sub	$x - y$	整数减法	2-补码
neg	$-y$	算数求反	2-补码
eq	若 $x=y$ 则为真，否则为假	相等判断	
gt	若 $x>y$ 则为真，否则为假	大于判断	
lt	若 $x<y$ 则为真，否则为假	小于判断	
and	*x* And *y*	按位“与”操作	
or	*x* Or *y*	按位“或”操作	
not	Not *y*	按位“非”操作	

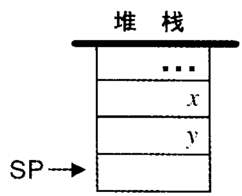


图 7.5　算术和逻辑堆栈命令

在本章的前面介绍中，内存访问命令使用伪命令 *pop* 和 *push x* 来表示，这里符号 *x* 代表在某个全局内存中的一个独立的存储单元。然而正规来说，VM 操纵 8 个独立的**虚拟内存段**（*virtual memory segments*），列在图 7.6 中。

内存访问命令　所有的内存段都通过相同的两个命令来进行存取：

- push *segment index*　　将 *segment*[*index*]的值压入堆栈。
- pop *segment index*　　将栈顶元素弹出然后存入 *segment*[*index*]。

段　名	功　能	说　明
argument	存储函数的参数	当进入函数时，由 VM 实现机制进行动态分配
local	存储函数的局部变量	当进入函数时，由 VM 实现机制进行动态分配并被初始化为 0
static	存储同一.vm 文件中所有函数共享的静态变量	由 VM 实现机制为每个.vm 文件进行分配；被.vm 文件中的所有函数共用
constant	包含所有常数的伪段（Pseudo-segment），常数的范围为 0...32767	由 VM 实现机制来模拟；该段对于程序中所有函数都是可见的
this that	通用段，能够与堆中不同区域相对应来满足各种程序编写的需求	任何 VM 函数可以使用这两个段来操纵堆中指定的区域
pointer	该段由 2 个内存单元组成，用来保存 this 和 that 段的基地址	任何 VM 函数可以将 pointer 0（或 1）设置到某一地址上；这相当于将 this（或 that）段联结到起始于该地址的堆区域上
temp	固定的段，由 8 个内存单元组成，用来保存临时变量	被任何 VM 函数用于任何用途。被程序中的所有函数共享

图 7.6　每个 VM 函数使用的内存段

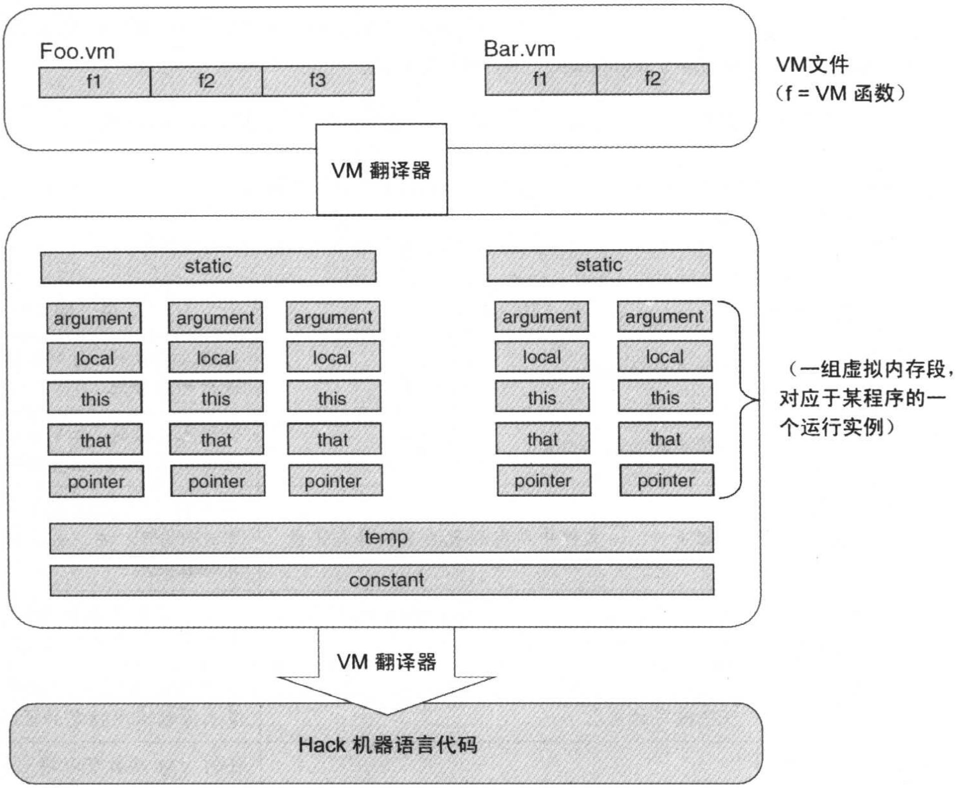


图 7.7 VM 实现机制所维护的虚拟内存段

以上命令中的 *segment* 代表图 7.6 中 8 个虚拟内存段之一，*index* 是非负的整数。比如，`push argument 2` 和 `pop local 1` 语句将会把函数的第三个参数的值存储在函数的第二个局部变量中（每个 segment 的 index 均从 0 开始）。

VM 文件、VM 函数和它们各自的虚拟内存段之间的关系在图 7.7 中有明确描述。

除了这 8 个虚拟内存段可被 VM 的 pop 和 push 直接操作，VM 还操纵两个暗含的数据结构称为**堆栈**（*stack*）和**堆**（*heap*）。这些数据结构从来不会被直接提到，但随着 VM 对它们进行操作，它们的状态却会在后台变化。

堆栈（stack） 考虑两条连续的命令语句 `push argument 2` 和 `pop local 1`，这个在前面提及过。这样的 VM 操作的工作内存就是**堆栈**。数据值并不是简单地直接从一个单元跳到另一个单元，而是必须经过堆栈中转。尽管堆栈是 VM 结构中的核心角色，但是在 VM 语言中从未显式地体现出它的功能。

堆（heap） 处在 VM 后端的另一个数据结构就是**堆**。堆是 RAM 区域的名字，用来存储对象和数组数据。这些对象和数组能够通过 VM 命令来操纵，我们马上将会看到。

7.2.4 程序流程控制命令和函数调用命令 Program Flow and Function Calling Commands

VM 提供 6 个附加的命令，将会在下一章中详细地讨论。为了完整起见，这些命令还是在下面列了出来。

程序流程控制命令：

```
label symbol          //标签声明
goto symbol           //无条件分支
if-goto symbol        //条件分支
```

函数调用命令：

```
function 函数名 nLocals    //函数声明，指明函数的本地变量个数
call 函数名 nArgs          //调用函数，指明函数的参数的个数
return                     //将程序控制权返回给调用者
```

（在这些命令中，*函数名*是字符串，*nLocals* 和 *nArgs* 都是非负整数。）

7.2.5 Jack-VM-Hack 平台中的程序元素 Program Elements in the Jack-VM-Hack Platform

让我们采取自顶向下的视角，审视典型的高级程序在完整编译过程中所涉及的所有程序元素，以此来结束对 VM 规范第一部分的阐述。在图 7.8 的顶部我们看见一个 Jack 程序（Jack 是简单的类 Java 语言，会在第 9 章中进行介绍）。每个 Jack 类包含一个或多个函数。

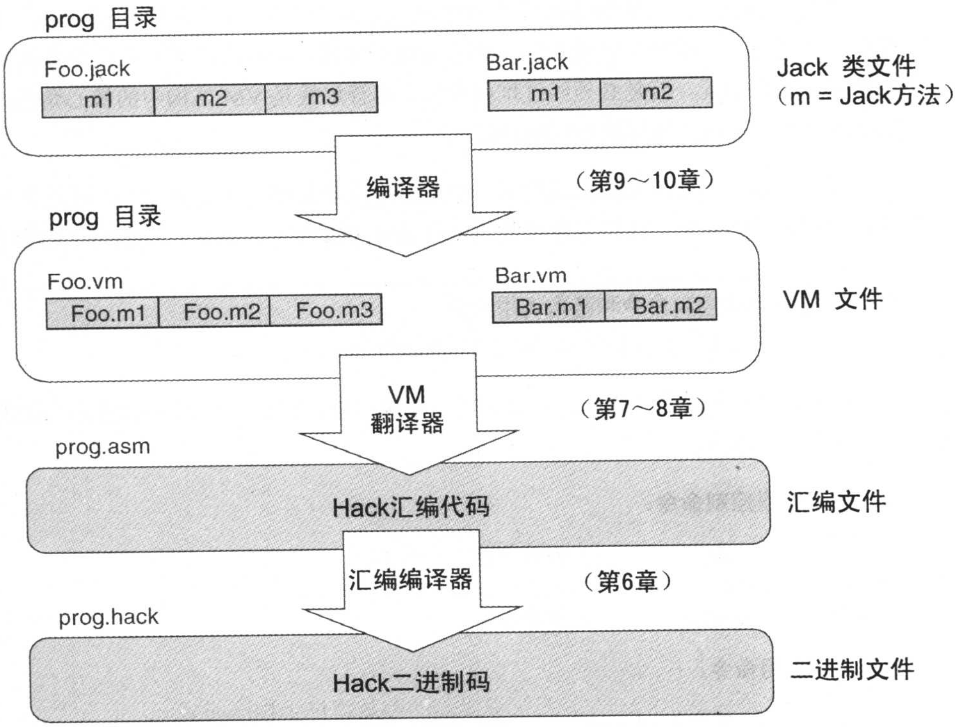


图 7.8 Jack-VM-Hack 平台中的程序要素

当 Jack 编译器对某目录下的 *n* 个`.jack`类文件（class file）进行编译时，它会生成 *n* 个对应的 VM 文件（保存在相同路径下）由两个类组成。每个在`.jack`文件`Yyy`下的 Jack 函数`xxx`被翻译成名为`Yyy.xxx`的 VM 函数，保存在对应的 VM 文件里面。

接下来，图中显示了 **VM 翻译器**（*translator*）是如何对该目录下的 *n* 个 VM 文件进行处理，进而组合生成一个单一的汇编程序的。汇编程序主要处理两件事情。首先，它对每个 VM 函数和文件的虚拟内存单元以及隐含的堆栈进行仿真。然后，它在目标平台上执行 VM 命令。这是通过使用机器语言指令（由 VM 命令翻译而来）操纵仿真的 VM 数据结构来实现的。如果所有部分工作正常，即编译器和 VM 翻译器和汇编程序都得到了正确的执行，那么目标平台最终就会执行原始 Jack 程序所交付的任务。

7.2.6 VM 编程实例

VM Programming Examples

下面来讨论如何用 VM 抽象来表达高级程序中典型的编程任务，以此结束本部分的阐述。我们给出了 3 个例子：（1）典型的算法任务；（2）典型的数组处理；（3）典型的对象处理。本节这 3 个例子均与 VM 实现无关，实际上可跳过不看，并不影响阅读。

本节的主要目的是为了给大家介绍编译器是如何利用 VM 抽象将高级程序翻译成 VM 代码的。事实上，VM 程序很少由程序员来编写，更多地是由编译器来完成此工作。因此，我们对每个例子给出高级语言代码片断，然后显示出使用 VM 代码的等价表示。这里使用 C 语言风格的语法来表示高级语言。

典型的算法任务 在图 7.9 的上部给出了乘法算法的源代码。我们（更准确地说应该是编译器）应该如何用 VM 语言来表达这个算法呢？首先，高级语言结构比如 `for` 和 `while` 必须使用 VM 简单的“goto 逻辑”来表示。同样地，高级语言的算术操作和逻辑操作必须使用面向堆栈的命令来重写。相应的代码如图 7.9 所示（`function`，`label`，`goto`，`if-goto`，`return` 这些 VM 命令的准确语义在第 8 章中有明确阐述，其实它们的意思是显而易见的）。

下面主要来看看图 7.9 下部描述的虚拟语言片段。可以看到当 VM 函数开始运行时，它假定：（1）堆栈是空的；（2）用来被操作的参数值位于程序的 `argument` 部分；（3）在程序 `local` 部分的局部变量都被初始化为 0。

接下来看看该算法的 VM 表示。前面说过，VM 命令不能使用符号化的参数和变量名，只能按照<*segment index*>的形式来进行引用。然而，从前者翻译到后者是很简单的。所需要做的只是将 `x,y,sum` 和 `j` 分别映射成 `argument 0`、`argument 1`、`local 0` 和 `local 1`，然后将代码中所有参数和变量名的符号都替换为各自对应<*segment index*>形式。

综上所述，在 VM 函数开始执行时，假定其完全被封闭在一个“与世隔绝的自在世界”当中（该世界只包括初始化了的 `argument` 和 `local` 部分以及一个空堆栈），等待有命令对其进行操作。VM 实现正是负责在 VM 函数执行之前，为 VM 函数营造这个虚拟世界观，这将在下一章中详细介绍。

高级代码（C 语言风格）

```
int mult(int x, int y) {
  int sum;
  sum = 0;
  for(int j = y; j != 0; j--)
      sum += x; // 重复的相加操作
  return sum;
}
```

初步近似

```
function mult
  args x, y
  vars sum, j
  sum = 0
  j = y
loop:
  if j = 0 goto end
  sum = sum + x
  j = j - 1
  goto loop
end:
  return sum
```

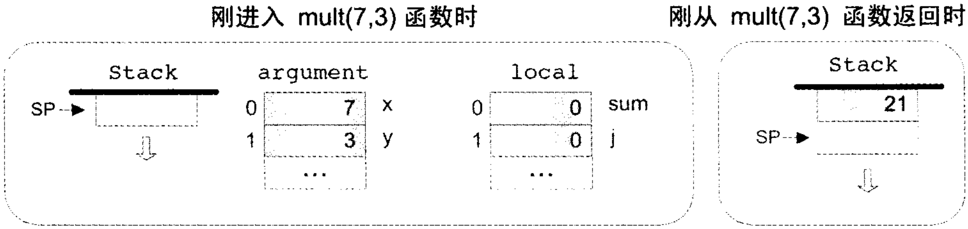
伪 VM 代码

```
function
mult(x,y)
  push 0
  pop sum
  push y
  pop j
label loop
  push 0
  push j
  eq
  if-goto end
  push sum
  push x
  add
  pop sum
  push j
  push 1
  sub
  pop j
  goto loop
label end
  push sum
  return
```

最终的 VM 代码

```
function mult 2   //2 个局部变量
  push constant 0
  pop local 0      //sum=0
  push argument 1
  pop local 1      //j=y
label loop
  push constant 0
  push local 1
  Eq
  if-goto end // if j=0 goto end
  push local 0
  push argument 0
  Add
  pop local 0      //sum=sum+x
  push local 1
  push constant 1
  Sub
  pop local 1      //j=j-1
  goto loop
label end
  push local 0
  return           //返回 sum
```

图 7.9 VM 编程范例



续图 7.9　VM 编程范例

数组处理　数组实际上就是一组有索引的对象集合。假设高级语言程序创建了名为 bar 的由 10 个整数组成的数组，然后填入了 10 个数字。现在假定数组的基地址已经被映射到 RAM 地址 4315 处。假设现在程序想要执行命令 bar[2]=19 。那么如何来在 VM 层级上实现这个操作呢？

在 C 语言里面，这样的操作可以被描述为*(bar+2)=19 ，意思是**“将 RAM 中地址为** bar+2 **的位置赋予值** 19” [1]。如图 7.10 所示，这个操作能在 VM 中很好地实现。

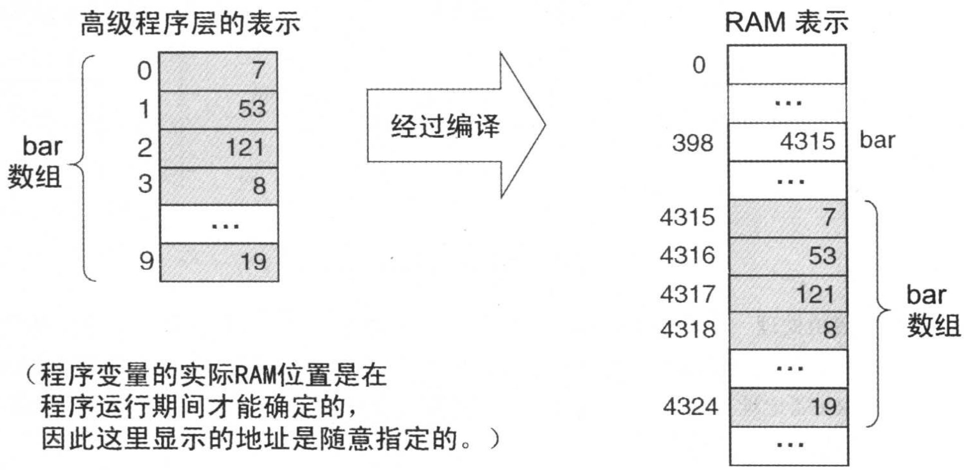
那么，高级语言命令（比如 bar[2]=19）首先是如何被翻译成图 7.10 所示的 VM 代码的呢？这个转化过程在 11.1.1 小节中有介绍，那里将讨论编译器的代码生成特性。

对象处理　高级语言的程序员将对象看成是封装了数据（由成员**字段**即 *fields*；或者**属性**即 *properties* 组成）和相关代码（由**方法**即 *methods*，组成）的实体。然而从本质上讲，每个对象实例（object instance）的数据是在 RAM 上被序列化（serialized）成一串数字，这串数字代表对象中各个字段的值。因此，对象的低级处理和数组的低级处理很相似。

例如，考虑一个动画程序，该程序在屏幕上弹球。假设每个球对象被整数字段 x、y、radius 和 color 来描述，而且程序创建了该对象并命名为 b 。那么在计算机里这个对象的内部形式将是什么样呢？

像所有其他对象实例一样，它会被存储在 RAM 里。当程序在创建一个新对象，编译器都会以字为单位计算对象的大小，然后操作系统会寻找一个足够大的 RAM 空间并分配

[1] VM 中有两个指针 pointer0 和 pointer1，分别对应两个虚拟内存段 this 和 that；如果将 pointer0 或 pointer1 赋予某特定内存地址 *address*，那么实际上就是将对应的 this 和 that 两个虚拟内存段映射到内存中以 *address* 地址开始的一块内存段。——审校者




```
VM 代码
// 假设 bar 数组是该高级程序中声明的第一个本地变量。
// 下面的 VM 代码执行操作 bar[2]=19，也就是，*(bar+2)=19。
push local 0          // 获取 bar 的基地址
push constant 2
add
pop pointer 1         // 将 that 基地址设为(bar+2)
push constant 19
pop that 0            // *(bar+2)=19
...
```

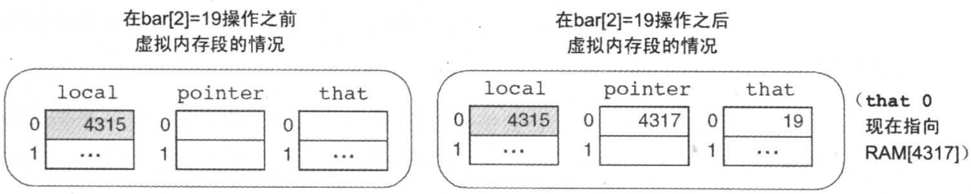


图 7.10 应用 pointer 和 that 段对 VM 的数组进行操作

给该对象来存储它的内容（有关这个操作的细节会在第 11 章中介绍）。现在，假设 b 对象已经被分配了从 3012 地址到 3015 地址的 RAM 空间，如图 7.11 所示。

假设现在有一高级语言程序中有名为 `resize` 的方法，以 `Ball` 对象和整数 `r` 为参数，该方法将球的半径设为 `r` 。这个逻辑的 VM 表示如图 7.11 所示。

将 `pointer 0` 的值设为 `argument 0`，实际上就好似将 `this` 虚拟内存段的地址设成对象的基地址，这样通过 VM 指令就可以用 `this` 段和一个索引号来访问对象中任何数据成员。请注意，使用的索引号是我们所访问的数据成员地址相对于其对象基址的偏移量。

但是编译器是如何将 `b.radius=17` 翻译成图 7.11 中所示的 VM 代码的呢？编译器是如何知道对象的 *radius* 字段在实际的表示中和第三个字段相关的呢？要想解决这些问题，还得参看 11.1.1 小节。

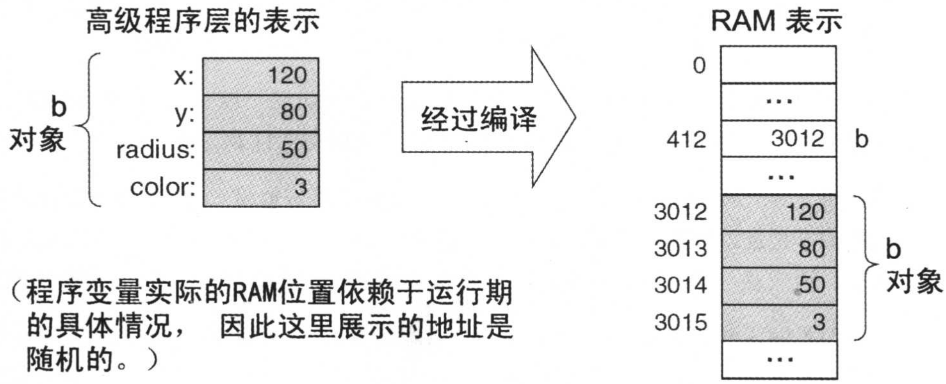
7.3 实现 Implementation

介绍到这里，虚拟机还是一个抽象的工件（artifact）。如果想真正地使用它，就必须在真实的平台上实现它。构建这样的 VM，从概念上包含两个任务。首先必须在目标平台上对 VM 进行仿真。特别是，VM 规范中提到的每个数据结构，也就是堆栈和虚拟内存段，都必须通过某种方式在目标平台上表示出来。其次，每个 VM 命令都必须被翻译成一系列具有目标平台语意的指令。

这部分介绍如何将 7.2 节中讨论的 VM 描述在 Hack 平台实现。首先得定义从 VM 要素和操作到 Hack 硬件和机器语言之间的“标准映射”。然后，提出实现这种映射的软件设计思路。接着会交替地使用 VM **实现**（*implementation*）或者 VM **翻译器**（*translator*）这两个名词来指代这个软件。

7.3.1 Hack 平台上的标准 VM 映射，第 I 部分 Standard VM Mapping on the Hack Platform, Part I

如果你重新去看虚拟机规范这一部分，你会发现里面没有提到任何关于 VM 赖以实现的硬件平台体系结构的规范要求。对于虚拟机而言，平台独立性至关重要：



VM 代码

```
// 假设 b 对象和整数 r 被作为函数的前两个参数传入。
// 下列代码实现了 b.radius=r 操作。
push argument 0     // 取得 b 的基地址
pop pointer 0       // 将 this 段指向 b
push argument 1     // 取得 r 的值
pop this 2          // 将 b 的第 3 个字段设置为 r
...
```

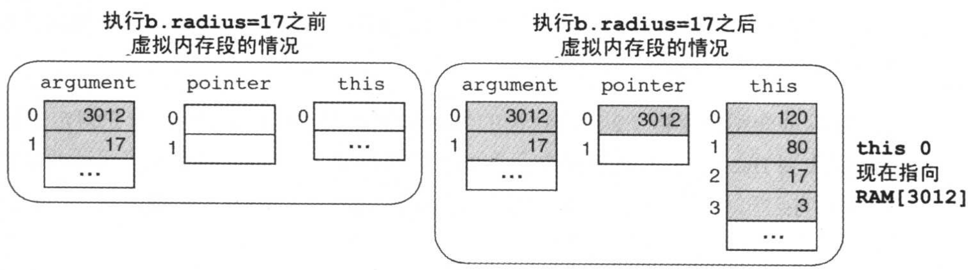


图 7.11 应用 pointer 和 this 段对 VM 的对象进行操作

你不想让其依赖于任何一种特定的硬件平台，因为你希望虚拟机能够潜在地适合于在所有硬件平台上运行（包括那些还未构建的硬件平台）。

VM 设计者应该允许程序员用任何他们觉得合适的途径去在目标机上实现 VM。然而，我们建议将一些关于 VM 应该怎样映射到目标平台的设计指南提供给设计人员，而不是去把这些决策性的东西全都交给设计者来决定。这些指导性的原则，称为**标准映射**（*standard mapping*）。为什么要提供它们呢？有两个原因。首先，它们制定了公共的约定，以此来规范基于 VM 的程序如何与没有使用到 VM 的编译器（比如直接产生二进制代码的编译器）生成的程序之间进行交互。其次，我们希望 VM 的开发人员能够运行标准化的测试，即符合标准映射的测试。这样，测试程序和软件能够由不同的人员来编写，这是软件工程中通常推荐的方式。因此本节剩余的内容将会详细介绍 VM 在我们熟悉的硬件平台即 Hack 计算机上的标准映射。

从 VM 到 Hack 的翻译　前面介绍过 VM 程序是由一个或多个扩展名为`.vm`的文件构成的集合，每个文件包含一个或多个 VM 函数，每个函数又由一系列 VM 命令构成。VM 翻译器将这些`.vm`文件作为输入，产生一个单一的 Hack 汇编语言`.asm`文件作为输出（见图 7.7）。每个 VM 命令被翻译成 Hack 汇编代码。`.vm`文件中函数的顺序对程序的执行没有影响。

RAM 用法　Hack 计算机的数据内存由 32K 个 16-位字组成。前 16K 作为通用 RAM。下一个 16K 包含 I/O 设备的内存映像。VM 实现应该使用如下的间隔分配方式：

RAM 地址	功　用
0 – 15	16 个虚拟寄存器，用法详见后述
16 – 255	VM 程序的所有 VM 函数的静态变量
256 – 2047	栈
2048 – 16383	堆（用于存放对象和数组）
16384 – 24575	内存映像 I/O

前面介绍过，根据 **Hack 机器语言规范**，任何汇编程序都能够使用`R0`～`R15`的符号来指代 0～15 的 RAM 地址单元。另外，规范还规定了汇编程序可以使用符号`SP`、`LCL`、`ARG`、

THIS 和 THAT 来分别指代 RAM 地址 0～4（也就是 R0～R4）。在汇编语言中规定这种表示法是富有远见的做法，目的是为了加强 VM 实现的可读性。在 VM 程序里面的这些寄存器的用法如下所示：

寄存器	名 称	功 用
RAM[0]	SP	栈指针：指向栈中下一个最顶位置
RAM[1]	LCL	指向当前 VM 函数 local 段的基址
RAM[2]	ARG	指向当前 VM 函数 argument 段的基址
RAM[3]	THIS	指向当前 this 段（在堆中）的基址
RAM[4]	THAT	指向当前 that 段（在堆中）的基址
RAM[5-12]		保存 temp 段的内容
RAM[13-15]		可被 VM 实现用作通用寄存器

内存段映射

local, argument, this, that ：每一个这样的段都被直接映射到 RAM；通过专用寄存器（分别为 LCL、ARG、THIS、THAT）来保存，其物理基地址，就可以维持其在 RAM 中的位置。如此一来，对这些段的第 *i* 个数据项的访问，应该被翻译成“获取 RAM 中地址为（*base*+*i*）的值的汇编代码，这里 *base* 是存储在各段专有寄存器中的当前值。

pointer, temp ：这些段被直接映射在 RAM 中的一个固定区域上。pointer 段被映射在 RAM 位置 3～4（也称为 THIS 和 THAT）上，temp 段被映射在 RAM 位置 5～12（也称为 R5, R6,…, R12）上。因此访问 point i 应该被翻译成访问 RAM 位置 3+*i* 的汇编代码，访问 temp i 应该被翻译成访问 RAM 位置 5+*i* 的汇编代码。

Constant ：这个段是真正虚拟的，因为它不占用目标平台上的任何物理存储空间。VM 实现通过简单地提供常数 *i* 来处理任何 VM 对 *<constant i>* 的访问。

Static ：根据 Hack 机器语言规范，在汇编程序中每遇到一个新的符号时，编译器就为其分配一个新的 RAM 单元，从地址 16 处开始。利用这个规定，使用汇编语言符号 f.j 来表示 VM 文件 f 中的每个静态变量数字 j。例如，假设文件 Xxx.vm 包含命令 push static 3。该命令能被翻译成 Hack 汇编命令@Xxx.3 和 D=M ，接下来的汇编代码将 D 的值压入堆栈。该 static 段的实现方式有点机巧诡异，但确实起作用。

汇编语言符号 下面重新给出了所有在 VM 实现机制中使用的汇编语言符号，它们符合标准映射规范。

符 号	功 用
SP, LCL, ARG, THIS, THAT	这些预定义的符号分别指向栈顶以及 local, argument, this, that 各段的基址
R13-R15	这些预定义符号可用于任何目的，是通用的
Xxx.j 符号	文件 Xxx.vm 中的每个静态变量 j 被翻译为汇编符号 Xxx.j。在后续汇编过程中，这些符号化变量将会由 Hack 汇编编译器为其分配 RAM 空间
控制流符号	VM 命令 function, call, label 的实现涉及到生成特殊的标签符号，将在第 8 章详述

7.3.2 关于 VM 设计实现的建议
Design Suggestion for the VM Implementation

VM 翻译器应该接收一个简单的单一命令行参数，如下：

```
Prompt> Vmtranslator source
```

这里 *source* 是 Xxx.vm 文件的文件名（必须有扩展名）或者是包含一个或多个 .vm 文件的路径名称（不能带扩展名）。翻译的结果通常是生成一个单一的名为 Xxx.asm 的汇编语言文件，保存在与 Xxx.vm 相同的路径下。翻译的代码必须符合 Hack 平台上的标准 VM 映射规则。

7.3.3 程序结构
Program Structure

本书计划采用一个主程序和两个模块：*parser* 和 *codewriter* 来实现 VM 翻译器（即 VM 实现机制）。

Parser 模块

Parser ：分析 .vm 文件，封装对输入代码的访问。它读取 VM 命令并解析，然后为它们的各个部分提供方便的访问入口。除此之外，它还移除代码中所有的空格和注释。

程　序	参　数	返回值	功　能
构造函数	输入文件/输入流	—	打开输入文件/输入流，准备进行语法解析
hasMoreCommands	—	Boolean	输入当中还有更多命令吗
advance	—	—	从输入读取下一条命令，将其指定为“当前命令”。仅当 hasMoreCommands()返回为真时，才能调用此程序。初始情况下，没有“当前命令”
commandType	—	C_ARITHMETIC，C_PUSH，C_POP，C_LABEL，C_GOTO，C_IF，C_FUNCTION，C_RETURN，C_CALL	返回当前 VM 命令的类型。对于所有算术命令，总是返回 C_ARITHMETIC
arg1	—	字符串	返回当前命令的第一个参数。如果当前命令类型为 C_ARITHMETIC，则返回命令本身（如 add，sub 等）。当前命令类型为 C_RETURN 时，不应该调用本程序

续表

程 序	参 数	返回值	功 能
Arg2	—	int	返回当前命令的第二个参数。仅当当前命令类型为 C_PUSH, C_POP, C_FUNCTION, C_CALL 时，才可调用

CodeWriter **模块**

CodeWriter ：将 VM 命令翻译成 Hack 汇编代码。

程 序	参 数	返回值	功 能
构造函数	输出文件/输出流	—	打开输出文件/输出流，准备进行写入
setFileName	filename(字符串)	—	通知代码写入程序，新的 VM 文件翻译过程已经开始
writeArithmetic	Command(字符串)	—	将给定的算术操作所对应的汇编代码写至输出
WritePushPop	Command(C_PUSH 或 C_POP), segment(字符串), index(int)	—	将给定的 command（命令类型为 C_PUSH 或 C_POP）所对应的汇编代码写入至输出
Close	—	—	关闭输出文件
说明：第 8 章将会介绍本模块中更多的程序			

主程序　主程序应该构造一个 Parser 和一个 CodeWriter ；Parser 用来解析 VM 输入文件；CodeWriter 用来将生成的 hack 汇编代码写入相应的输出文件.asm 文件中。 另

外，主程序应该读取输入文件中的每一条命令并为其生成对应的汇编代码。

如果程序的参数是路径名称而不是文件名称，那么主程序应该处理该路径下的所有 `.vm` 文件。那么，它应该使用独立的 `Parser` 来处理每个输入文件，以及单一的 `CodeWriter` 来处理输出。

7.4 观点 Perspective

本章开发了用于高级语言的编译器。遵循现代软件工程实践的经验，采用了两层（two-tier）编译模型来进行开发。在**前端**（*frontend*）层级（第 10 章和第 11 章的内容），高级代码被翻译为运行在虚拟机上的中间代码。在**后端**（*backend*）层级（本章以及下一章的内容），中间代码被翻译为目标硬件平台的机器语言（如图 7.1 和图 7.9 所示）。

将中间代码作为显式的虚拟机语言，这种想法可以追溯到 20 世纪 70 年代，被当时的几种 Pascal 编译器广为采用。这些编译器生成的中间代码称为“p-code（p-代码）”，能在任何实现它的计算上运行。随着 20 世纪 90 年代中期互联网（World Wide Web）的广泛普及应用，跨平台兼容性变成了令人头疼的问题。为了解决这个问题，Sun Microsystems 公司开发了一种新的编程语言，使其能够在任何联网的计算机以及数字设备上运行。基于此初衷而诞生的 Java 语言正是建立在称为 **JVM**（*Java Virtual Machine*）的中间代码执行模型之基础上的。

JVM 是一种规范，描述了 Java 编译器的目标语言，即称为 *bytecode* 的中间语言。用 bytecode 编写的文件被作为 Java 程序的动态代码发布版本在互联网上进行发布，最常用的是嵌在网页里面的 applets（Java 小应用程序）。当然，为了运行这些程序，客户端计算机必须安装相应的 JVM 实现。这种 JVM 实现也称为 *Java Run-time Environment*（JREs，Java 运行时环境），在许多“处理器/OS”平台（包括游戏机和手机）上被广为采用。

21 世纪头十年之初，微软（Microsoft）公司通过推出.NET 体系架构加入了此领域。.NET 的核心是称为 *Common Language Runtime*（CLR，公共语言运行时）的虚拟机模型。根据微软对.NET 的技术展望，许多语言（包括 C++、C#、Visual Basic，以及 Java 的变体 J#）

都能够被编译为运行在 CLR 上的中间代码。这样一来，用不同语言编写的代码就可以在同一个公共运行时环境的基础上进行交互并分享程序库。

需要注意的是，在虚拟机模型发挥互操作性的巨大潜力之前，必须要将一个关键成分加入虚拟机模型，这个成分就是通用软件程序库（common software library）。Java 虚拟机带有 Java 程序库，微软的虚拟机带有 CLR。可以将这些软件程序库看作小型操作系统，为运行在虚拟机上的语言提供了统一的服务，比如内存管理、GUI 组件、字符串处理函数、数学函数等等。本书第 12 章描述一种这样的程序库。

7.5 项目 Project

本节阐述如何构建本章所描述的 VM 翻译器。下一章会将本章的翻译器作进一步的功能扩展，最终形成完整的虚拟机实现。开始之前，需要说明两点：第一，7.2.6 节的内容与本项目无关；第二，VM 翻译器是用来生成 Hack 汇编代码的，因此你可能有必要复习一下关于 Hack 汇编语言的内容（第 4.2 节）。

目标 构建 VM 翻译器的第一个部分（第二部分将在项目 8 中实现），主要包括堆栈运算和 VM 语言的内存访问命令。

资源 需要两种工具：实现 VM 翻译器所需要的编程语言，以及与本书配套的 CPU 仿真器。该 CPU 仿真器可以执行 VM 翻译器生成的机器代码——这是一种测试 VM 翻译器的间接方法。另一个好用的工具是与本书配套的可视化 VM 仿真器。可以在构建自己的 VM 实现之前，使用可视化 VM 仿真器来了解 VM 实现的工作原理。关于可视化仿真器的更多信息，请参考 VM 仿真器教程。

约定 按照 VM 规范第 I 部分（7.2 节）以及 Hack 平台标准 VM 映射第 I 部分（7.3.1 节）的描述，编写 VM-to-Hack 的翻译器。然后使用该翻译器来翻译这里提供的 VM 程序，

产生对应的 Hack 汇编语言程序。用 CPU 仿真器执行这些生成出来的汇编程序，应该符合测试脚本以及比较文件所要求的结果。

建议的实现步骤

建议采用两个阶段来完成构建。这样你就能以递增的方式，利用提供的测试程序对你的实现进行单元测试。

第一阶段：堆栈运算命令 VM 翻译器的首版应该实现 VM 语言的 9 个堆栈运算命令和逻辑命令，以及 `push constant` *x* 命令（该命令有助于测试实现的 9 个命令）。注意，`push constant` *x* 命令是通用 `push` 命令的特化版本：其中第一个参数是常量，第二个参数是十进制常量。

第二阶段：内存访问命令 翻译器的第二版应该包含 VM 语言中 `push` 和 `pop` 命令的完整实现，能够处理所有 8 个内存段。建议将本阶段细分为如下几个步骤：

0. 已经能够处理 `constant` 段；

1. 然后，处理 `local` 段、`argument` 段、`this` 段、`that` 段；

2. 然后，处理 `pointer` 段和 `temp` 段，即要能够允许修改 `this` 段和 `that` 段的基地址；

3. 最后，处理 `static` 段。

测试程序

下列 5 个 VM 程序分别适用于在上述实现各阶段进行单元测试。

第一阶段：堆栈运算

- `SimpleAdd`：压入并相加两个常量。
- `StackTest`：执行一系列堆栈上的运算和逻辑操作。

第二阶段：内存访问

- `BasicTest` ：使用虚拟内存段执行 pop 和 push 操作。
- `PointerTest` ：使用 pointer 段、this 段、that 段执行 pop 和 push 操作。
- `StaticTest` ：使用 static 段执行 pop 和 push 操作。

对于每个程序 Xxx，我们提供了 4 个文件。首先是程序代码文件 Xxx.vm 。其次是脚本文件 XxxVME.tst ，让程序能够在 VM 仿真器中运行，以便让你了解程序的测试目的。使用你编写的 VM 翻译器将程序翻译之后，使用提供的 Xxx.tst 文件和 Xxx.cmp 文件在 CPU 仿真器上对翻译出来的汇编代码进行测试。

提示

初始化 为了运行翻译过的 VM 程序，VM 程序必须包含一段启动代码，以便让 VM 实现在宿主平台上启动执行。另外，为了保证 VM 代码正常运作，VM 实现必须虚拟内存段的基地址映射到选定的 RAM 位置。由于启动代码和虚拟内存段的初始化这两件事情在下一个项目中才会完成，因此这里遇到的困难是我们需要现成可用的初始化操作，以便能够执行本项目的测试程序。好在你不必担心这个问题，因为配套提供的测试脚本已经包含了所有必要的初始化操作（这是专门为此项目而编写的）。

测试和调试 对于每个测试程序，都按照如下的步骤进行操作：

1. 在 VM 仿真器上运行 Xxx.vm 程序并使用 XxxVME.tst 测试脚本，以便熟悉测试程序的测试目的。

2. 使用部分实现的翻译器来翻译 .vm 文件。生成的结果应该是以 Hack 汇编语言编写的 .asm 程序的文本文件。

3. 检查测试翻译过来的 .asm 程序。如果存在可见的语法错误（或其他错误），请调试并修改你的翻译器。

4. 在 CPU 仿真器上使用提供的`.tst`和`.cmp`文件来运行翻译过来的`.asm`程序。如果有运行期错误，请调试并修改你的翻译器。

配套提供的测试程序是经过精心编写的，专门用于测试 VM 实现各个阶段的各种功能。因此，请务必按照我们建议的顺序和步骤来实现翻译器，并在各个阶段利用对应的测试程序进行测试。若没有按照顺序来实现，那么测试程序可能就失去测试作用了。

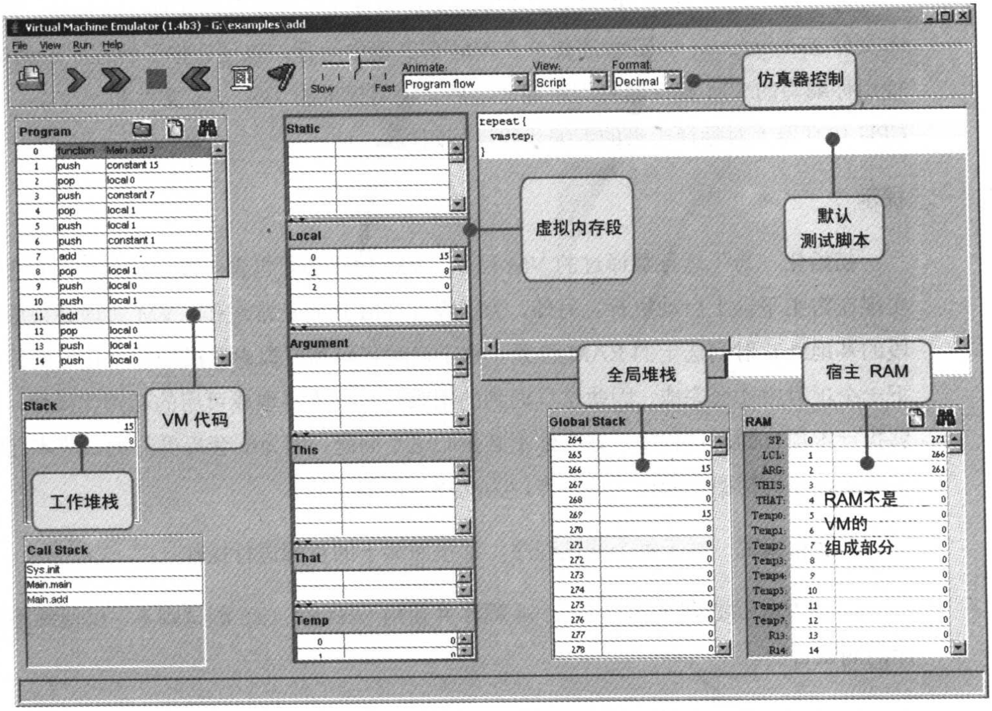


图 7.12 与本书配套的 VM 仿真器

工具

VM 仿真器 与本书配套的软件包内含了基于 Java 的 VM 实现。该 VM 仿真器允许直接执行 VM 程序而不必先将程序翻译为机器语言。这让你可以在实现翻译器之前，先熟悉 VM 环境。图 7.12 展示了使用中的 VM 仿真器的截图。

第 8 章　虚拟机 II：程序控制

Virtual Machine II: Program Control

If everything seems under control, you're just not going fast enough.

若一切看上去都还在掌控之中，那就说明你还跑得不够快。

——Mario Andretti（1940～），赛车冠军

第 7 章介绍了虚拟机（VM）的概念，最后在 Hack 平台上构建了基本的 VM。本章我们继续阐述 VM 抽象、语言和实现的开发。具体来说，我们要阐述如何设计基于堆栈的机器，以及用它来处理过程化（procedural）语言或者面向对象的（object-oriented）语言的嵌套子程序调用（嵌套子程序即：过程，procedures；函数，functions；方法，methods）。随着本章的介绍，我们将会扩展前面构建的基本 VM，最后开发出完整的 VM 翻译器。第 9 章中，在关于高级的基于对象的语言的介绍之后，该翻译器将会作为第 10 章和第 11 章中构建的编译器的后端程序（backend）。

在计算机科学领域的竞赛中，**堆栈处理**（*stack processing*）绝对是进入决赛的强大选手。前面的章节已介绍算术表达式和布尔表达式是如何利用基本堆栈操作来进行计算的。本章将继续介绍这个简单的数据结构如何支持像嵌套子程序调用、参数传递、递归和内存分配技术这样的复杂任务。大多数程序员不希望在这些功能上花功夫，而指望编译器来帮他们实现。现在我们就打开这个黑盒子，揭示这些基本编程机制是如何被基于堆栈的虚拟机实现的。

8.1 背景知识 Background

高级语言由高级表达式编写。比如，$x=-b+\sqrt{b^2-4\cdot a\cdot c}$ 可以用 `x=-b+sqrt(power(b,2)-4*a*c)` 来表示，它几乎是跟实际表达式描述得差不多。高级语言

支持这种表达式来源于它的三种语言原则。首先，高级语言允许在程序中根据需要自由定义如 sqrt 和 power 这样的高级操作（函数）。其次，高级语言允许自由使用（调用）这些子函数，就像使用一些如“+”和“*”这样的基本操作一样。最后，高级语言允许假设每个被调用的子程序都会被执行，并且会随着子程序的结束，程序的控制权顺利返回，例如返回给程序中调用语句的下一命令。流程控制命令（flow of control command）使得我们能更加自由地编写程序，比方说，`if ~(a=0) {x=(-b+sqrt(power(b,2)-4*a*c))/(2*a)} else {x=-c/b}`。

高级语言所具有的这种能将表达式自由组合的能力使得我们可以编写抽象代码，让我们能把精力主要放在算法的思想上，而不是机器的执行上。当然高级语言抽象级别越高，在底层要做的工作也越多。特别是，必须在底层控制**子程序**和**子程序调用者**（执行诸如 sqrt 和 power 的系统定义和用户定义操作的程序单元）之间的相互影响。对于在运行期的每个子程序调用，底层必须处理下面的一些细节。

- 将参数从调用者（caller）传递给被调用者（called subroutine）
- 在跳转并执行被调用者之前，先保存调用者的状态
- 为被调用者使用的局部变量分配空间
- 跳转并执行被调用者
- 将被调用者的运行结果返回给调用者
- 在从被调用者返回之前，回收其使用的内存空间
- 恢复调用者的状态
- 返回到调用语句之后的下一条语句继续执行

要考虑这些琐碎的事情是很头疼的，好在编译器把高级语言程序员从中解脱出来。那么编译器是如何做到这些的呢？如果我们用**堆栈机**（*stack machine*）来完成底层操作，那么以上工作的处理就很简单了，事实上堆栈结构本身的优势就在于处理这种类似任务。

本节剩余的内容将描述**程序控制流**（*program flow*）命令和**子程序调用**（*subroutine calling*）命令是如何在堆栈机上实现的。我们首先介绍程序控制流命令的实现（它十分简单，不需要内存管理），然后再来介绍复杂的子程序调用的实现。

8.1.1 程序控制流
Program Flow

计算机程序默认的执行顺序是线性的，即一个命令接着一个命令执行，这个连续的控制流偶尔被分支命令打断（比如在循环中执行迭代）。在底层编程中，分支逻辑利用 *goto destination* 命令实现，该命令让计算机跳转到程序中由目标参数（destination）指定的位置继续执行，而不是接着原指令的下一条指令线性执行。目的地址的指定方式可以有多种形式，最原始的一种就是指定即将执行的指令的物理地址。应用标签 *label* 来描述 jump 的目的地址可以建立稍微抽象一点的重定向命令（redirection command）。这种改进需要程序语言配有**标签指令**（*labeling directive*），它可将标签绑定到程序中指定位置。

这种基本的 *goto* 机制也可以很容易地实现条件分支。比如，*if-goto destination* 命令可以指示计算机仅在“给定的布尔条件为真”的情况下执行跳转；如果布尔条件为假，程序则继续线性地执行程序中的下一条命令。我们如何把布尔条件判断引入程序语言中呢？在堆栈机中，最常见的方法是根据堆栈栈顶元素的值来判断是否进行跳跃：如果它非 0，那么计算机就跳转到程序中指定的位置；否则就继续执行程序中的下一条命令。

在第 7 章中我们了解了 VM 的原始操作是如何计算任意给定的布尔表达式的，并将其结果置于栈顶的。这种布尔表达式与刚才介绍的 *goto* 和 *if-goto* 命令相结合，就可以表示任何编程语言中的任何控制结构的流程。图 8.1 给出两个典型的结构。

VM 命令 *label*、*goto label* 和 *if-goto label* 的底层实现是很容易的。所有的编程语言（包括“最”底层的）都有某种分支命令结构。例如，如果底层实现是基于将 VM 命令翻译成汇编代码，那么我们所要做的就是利用汇编语言的分支逻辑来重新表达 *goto* 命令。

8.1.2 子程序调用
Subroutine Calling

每种编程语言都有的特征是具有一组固定的内置命令集合。现代编程语言抽象机制的关键是允许程序员应用自定义的高级操作自由地扩展该语言的基本指令集。在过程化语言

控制流的结构	伪 VM 代码
if (条件) s1 else s2 ...	用于计算 ~(条件)的 VM 代码 if-goto L1 用于执行 s1 的 VM 代码 goto L2 label L1 用于执行 s2 的 VM 代码 label L2 ...
while (条件) s1 ...	label L1 用于计算 ~(cond)的 VM 代码 if-goto L2 用于执行 s1 的代码 goto L1 label L2 ...

图 8.1　goto 命令的底层控制流

（procedural language）中，高级操作称为**子程序**（*subroutines*）、**过程**（*procedures*）或者**函数**（*functions*）；在面向对象的语言中，它们通常称为**方法**（*methods*）。贯穿本章，所有这些高级程序单元都是指**子程序**（*subroutines*）。

在优秀的编程语言中，高级操作（利用一个子程序来实现）的使用与使用语言本身自带命令的感觉一样。例如，对于两个功能 *add* 和 *raise to a power*，大多数语言会把前者作为内置操作，后者可能会作为子程序来编写。尽管它们的实现是有区别的，但是对于调用者而言两个函数都应该以统一的方式被调用。这样就能让调用者使用统一的、可读性强的代码来方便地使用这两个操作。图 8.2 给出了遵循此原则的堆栈语言实现。

可以看到，调用内置命令和调用用户定义的子程序之间的唯一区别是用户自定义子程序之前要有关键字 call 。除此之外，内置命令和用户自定义子程序的实现过程都是一样的：两者都需要调用者将参数传递给被调用者，被调用者从堆栈中取出参数，在被调用者结束时将其处理结果压入堆栈返回给调用者。希望读者能从中体会到这种一致性的精妙之处。

```
// x+2
push x
push 2
add
...
```

```
// x^3
push x
push 3
call power
...
```

```
// (x^3+2)^y
push x
push 3
call power
push 2
add
push y
call power
...
```

```
// Power 函数
// result = 以第一个参数
// 为幂第二个参数为底的
// 计算结果
function power
// 此处代码省略
push result
return
```

图 8.2 子程序的调用。基本命令（比如 add）和高级函数调用（比如 power）在参数处理和返回值方面是一致的

子程序（比如 power）通常都会使用局部变量（local variables）进行临时存储。在子程序的“生命周期”中，也就是从子程序开始执行直到遇到 return 命令为止，必须为这些局部变量分配内存。在子程序返回时，这些被局部变量占用的内存将被释放。当允许子程序被任意嵌套时，子程序的调用机制会变得很复杂：一个子程序可以调用另一个子程序；不仅如此，被调用的子程序还可以进一步去调用其他子程序，如此嵌套。而且，还允许子程序递归地调用自己；每个递归调用必须独立于其他调用，并且维护其自身的局部变量和参数。那么如何才能实现这种嵌套机制以及相关的内存管理机制呢？

调用和返回逻辑的线性嵌套分层特点让这个繁琐任务变得容易管理。虽然子程序调用链可以具有任意深度，可以有任意的递归，但是在任何时刻，只有调用链顶部的子程序才能被执行，而处于调用链中其他子程序将一直等待到该子程序执行完毕为止才能继续执行。在子程序调用的实现过程中，这种**后进先出**（*Last-In-First-Out*，LIFO）的处理方式和堆栈数据结构的操作方式完全吻合。当子程序 xxx 调用子程序 yyy 时，可以将 xxx 的环境变量压入（保存到）堆栈中，然后转去执行 yyy 。当 yyy 返回时，再将 xxx 的环境变量从堆栈中弹出（恢复），如果没有特殊的情况出现就继续执行 xxx。图 8.3 介绍了这种执行模式。

我们在概念上使用**帧**（*frame*）来代表子程序的局部变量的集合，它包括子程序的参数、工作堆栈和运行过程中所使用的内存段。在第 7 章中，术语“**堆栈**（*stack*）”指的是 *pop*、

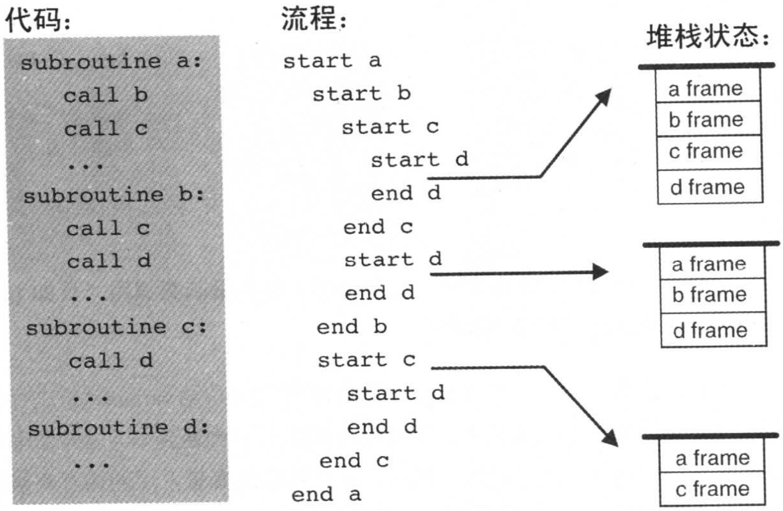


图 8.3 程序生命周期中具有代表性的三个调用点和对应的堆栈状态。全局堆栈中的所有帧都在等待当前帧的返回，当前帧返回之后，全局堆栈会变短，程序会继续操作紧挨着当前帧之后的帧（根据惯例，堆栈顶向下生长）

push、*add* 等操作的工作内存区。从现在起，我们说的"**堆栈**（*stack*）"是指**全局堆栈**，是由当前子程序的帧，和所有正在等待该子程序返回的其他子程序的帧所构成的内存区。这两个不同的堆栈概念是紧密相关的，因为当前子程序的工作堆栈位于全局堆栈的顶部。

在 `call xxx` 操作的底层实现中，先将调用者的帧保存到堆栈中，然后为子程序（`xxx`）的局部变量分配堆栈空间，最后跳到子程序 `xxx` 开始执行其代码。最后一步"大跳转"并不难实现。因为目标子程序的名字在 `call` 命令中已经被指定了，这样在调用过程中可以将这个目标程序名解析成一个内存地址，然后跳转到以该内存地址为基址的代码段（该代码段就是子程序的代码段）开始执行。而通过 `return` 命令从子程序中返回的过程却暗藏玄机，因为命令中并未指定返回地址。事实上，在所有调用子程序的过程中，调用者的返回地址都没有显式地给出。比如，因为像 `power(x, y)` 或 `sqrt(x)` 这样的子程序可以被**任何**调用者调用，那么它们的代码中就不可能给出具体的返回地址。所以，`return` 命令应该按如下方式来解释：它将程序的执行重定向到调用语句 `call` 命令的下一条命令所在的内存地址（不论这条命令的内存位置具体在哪），即称为**返回地址**（*return address*）。

基于堆栈调用的返回实现过程如图 8.3 所示。当我们用 `call xxx` 指令执行调用操作时，应该知道准确的返回地址：指令 `call xxx` 的下一条指令的内存地址。因此，我们将该指令内存地址作为返回地址压入堆栈保存，然后去执行子程序。当我们最终遇到 `return` 命令时，我们就将先前保存在堆栈中的返回地址弹出来，然后只用简单的 *goto* 命令跳转到这个地址就可以了。换句话说就是返回地址也可以保存在调用者的帧里面。

8.2 VM 规范详述，第 II 部分 VM Specification, Part II

第 7 章中介绍了基本的 VM 规范，本节将以**程序控制流**（*program flow*）和**函数调用**（*function call*）来对其进行扩展，以此完成全部 VM 规范的介绍。

8.2.1 程序控制流命令 Program Flow Commands

VM 语言有三种形式的程序控制流命令：

- `label` *label* 该命令标记程序中某条指令的位置，在程序中的跳转指令只能跳转到被 `label` *label* 命令所标示的位置，`label` 标签所指示的代码段范围就是程序中定义的函数体。*label* 可以是由任意字母、数字、下划线（_）、点（.）和冒号（:）组成的字符串，但不能以数字开头。
- `goto` *label* 该命令执行**无条件跳转**操作，使得程序跳转到 `label` *label* 命令标示的位置那里继续执行。跳转的目的地址必须位于同一个程序之内。
- `if-goto` *label* 该命令执行**条件跳转**操作。首先，将布尔表达式的运算结果从堆栈顶端弹出，如果该值非 0，那么程序就跳转到 *label* 标示的位置继续执行；否则，继续执行程序中的下一条命令。跳转的目的地址必须位于同一个函数内。

8.2.2 函数调用命令 Function Calling Commands

不同的高级语言对于程序单元（program units）概念采用不同的名称，包括**函数**（*function*）、**过程**（*procedure*）、**方法**（*method*）以及**子程序**（*subroutine*）。在整个编译模型（将在第 10 至 11 章中详细介绍）中，每个高级程序单元都被翻译成 VM **函数**，或简称为**函数**。

函数的名称是个全局量，可以是由任意字母、数字、下划线（_）、点（.）以及冒号（:）组成的字符串，但不能以数字开头。（如果在高级语言的 Foo 类中定义有函数 bar，那么该函数应该经过编译器翻译为对应的名为 Foo.bar 的函数）。函数名称的使用范围是全局的，即所有文件中的所有函数可通过这种全局名称相互调用。

VM 语言有三种函数相关的命令：

- `function` *f* *n* 一段函数名为 *f* 的代码，该函数有 *n* 个参数；
- `call` *f* *m* 调用函数 *f*，其 *m* 个参数已经被调用者压入堆栈；
- `return` 返回到调用者。

8.2.3 函数调用协议
The Function Calling Protocol

调用函数和从函数返回这两个操作可以从两个不同的角度来看，即调用者的角度和被调用者的角度。

调用者的角度

- 在调用函数之前，调用者必须将必要的参数压入堆栈；
- 接着，调用者使用 call 命令来调用函数；
- 被调用函数返回后，调用者先前压入堆栈的参数将被删除，并且函数的返回值将出现在栈顶；
- 被调用函数返回后，调用者的各内存段（如 argument、local、static、this、that 和 pointer）跟调用之前一样，temp 未被定义。

被调用者角度

- 当被调用的函数开始执行，其 argument segment 段被初始化为调用者所传递的参数，为其 local segment 段分配内存空间并初始化为 0，它的 static segment 段被置为其所属 vm 文件中的 static segment，工作堆栈为空。This、that、pointer 和 temp 四个指针均未初始化；
- 返回前，被调用函数必须将某个值压入堆栈。

在这里重复前面章节的一个观点：每当 VM 函数开始运行（或者继续前面的执行）时，要保证其处于自己的私有空间之内，这个私有空间是由各函数的虚拟内存和堆栈组成。VM 函数的指令就对其私有空间内的各个虚拟内存段和堆栈进行操作。VM 实现机制负责构建各 VM 函数的私有空间，这一点将在 8.3 节中详细介绍。

8.2.4 初始化
Initialization

VM 程序是一组相关的 VM 函数集合，一般来自于某种高级程序的编译。当 VM 实现开始运行（或者重启）时，按照惯例它总是执行名为 `Sys.init` 的无参数 VM 函数。接着该函数调用用户程序中的主函数。因此，生成 VM 的编译器必须保证每个翻译后的程序都有个这样的 `Sys.init` 函数。

8.3 实现
Implementation

这一节介绍了如何完成从第 7 章开始构建的 VM 实现机制，最终实现整个虚拟机。第 8.3.1 小节描述实现过程所必须的堆栈结构以及它在 Hack 平台上的标准映射。第 8.3.2 小节给出了范例，第 8.3.3 小节提供了设计建议以及实际构建 VM 实现的 API。

其中一些设计细节在技术上是相当复杂的，深究这些技术细节会使我们的注意力偏离整体的 VM 操作。第 8.3.2 小节展现了全局景观，介绍具体的 VM 实现过程。因此，读者在阅读 8.3.1 小节的过程中可以参考 8.3.2 小节。

8.3.1 Hack 平台上的标准 VM 映射，第 II 部分
Standard VM Mapping on the Hack Platform, Part II

全局堆栈　VM 的内存资源是通过维护一个全局的堆栈来得到的。每当调用一个函数时，该函数对应的帧（frame）就被压入全局堆栈。该帧包括被调用函数（called function）将要使用的参数；一组用于保存调用者状态的**指针**（*pointers*）；被调用函数的**局部变量**（被初始化为 0）；以及一个被调用函数将要使用的工作堆栈（当前为空）。图 8.4 给出了这个堆栈结构。

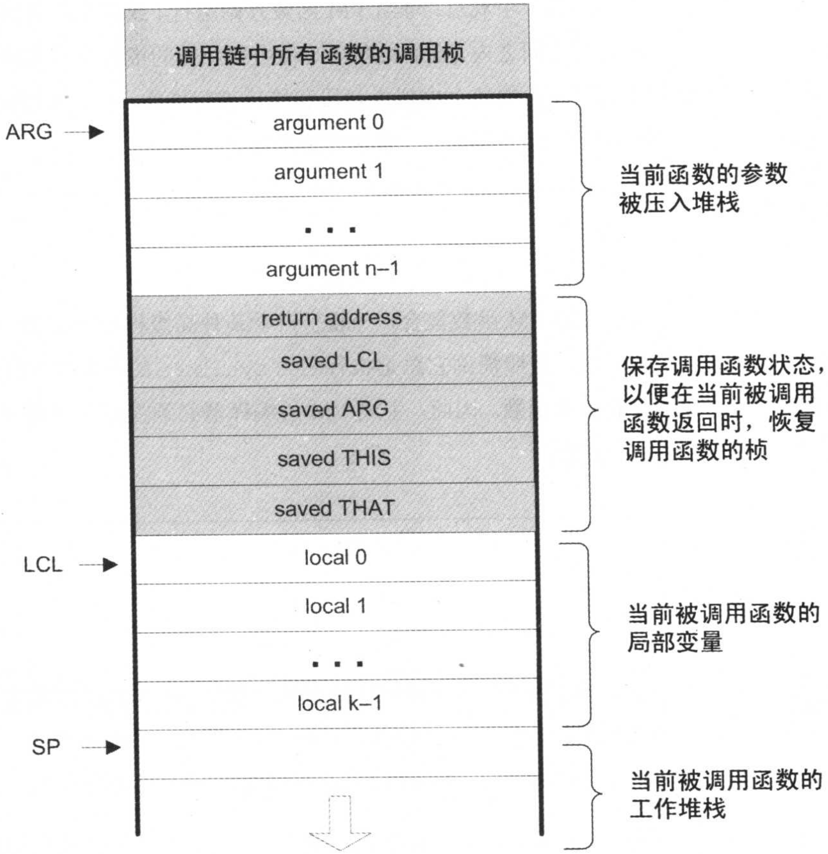


图 8.4　全局堆栈结构

要注意图 8.4 中的阴影区以及 ARG、LCL 和 SP 指针对于 VM 函数是不可见的（无法感知它们的存在）。这三个指针是更底层的 VM 实现在函数调用与返回协议中使用的。

我们如何才能在 Hack 平台上实现这种模型呢？前面介绍的标准映射协议指定了堆栈的内存地址应该为 256，意味着 VM 的实现机制可以从生成将 SP 指针设置为 256 的汇编代码（SP=256）开始，于是每当 VM 实现遇到诸如 pop、push、add 等命令时，就会生成

相应的汇编代码，这些代码通过操作 SP 和内存中对应的存储单元来实现对应的 pop、push、add 命令。所有这些工作已经在第 7 章中完成了。同样，当 VM 实现遇到诸如 call、function 和 return 之类的命令时，它也将生成维护图 8.4 中所示的堆栈结构的汇编代码。这个代码接下来会介绍。

函数调用协议的实现 函数调用协议及其对应的全局堆栈结构能够通过执行图 8.5 中给出的（用 Hack 汇编编写）伪指令在 Hack 平台上实现。

前面介绍过的 VM 实现是高级语言编写的**翻译器**（*translator*）程序。它以 VM 代码作为输入，然后生成汇编代码作为输出。

VM 命令	VM 实现生成的（伪）代码
call f n （在 n 个参数被压入堆栈之后调用函数 f）	`push return-address   // （使用下面声明的标签）` `push LCL              // 保存调用函数的 LCL 段指针` `push ARG              // 保存调用函数的 ARG 段指针` `push THIS             // 保存调用函数的 THIS 段指针` `push THAT             // 保存调用函数的 THAT 段指针` `ARG = SP-n-5          // 重置 ARG（n = 参数数量）` `LCL = SP              // 重置 LCL` `goto f                // 跳转控制` `(return-address)      // 为返回地址声明一个标签`
function f k （声明一个有 k 个局部变量的函数 f）	`(f)                   // 为函数入口声明一个标签` `repeat k times:       // k = 局部变量的个数` `PUSH 0                // 将它们全部初始化为 0`
Return （从当前函数返回）	`FRAME = LCL           // FRAME 是临时变量` `RET = *(FRAME-5)      // 将返回地址放入临时变量中` `*ARG = pop()          // 重置调用者的返回值` `SP = ARG+1            // 恢复调用者的 SP` `THAT = *(FRAME-1)     // 恢复调用者的 THAT 段指针` `THIS = *(FRAME-2)     // 恢复调用者的 THIS 段指针` `ARG = *(FRAME-3)      // 恢复调用者的 ARG 段指针` `LCL = *(FRAME-4)      // 恢复调用者的 LCL 段指针` `goto RET              // 跳转到返回地址（在调用者代码中）`

图 8.5 函数命令的 VM 实现。其中(返回地址)和(f)是符合 Hack 汇编语法习惯的标记声明

因此，图 8.5 中右栏描述的每个伪操作实际上是通过发出汇编语言指令实现的。注意，对于这样的“指令”，我们必须在生成的代码流中插入一些标记声明。

汇编语言符号　正如我们之前看到的，**程序控制流**（*program flow*）命令和**函数调用**（*function calling*）命令的实现需要 VM 实现在汇编语言层创建并使用特殊的符号（symbol）。这些符号列在图 8.6 中。为了表述的完整性，第 7 章中已经描述并实现了图表中的前三行。

符　号	用　法
SP, LCL, ARG, THIS, THAT	这些预定义符号分别指向栈顶和各虚拟段 local、argument、this、that 的基地址
R13-R15	这些预定义的符号可以用于任何用途
Xxx.j	在 VM 文件 Xxx.vm 中的每个静态变量 j 被翻译成汇编符号 Xxx.j 。在随后的汇编过程中，Hack 汇编编译器会为这些变量分配 RAM 空间
functionName$label	在 VM 函数 f 中的每个 label b 命令应该生成唯一的全局符号“f$b”，这里“f”是函数名，“b”是 VM 函数体内的标记符号。在将 VM 命令 goto b 和 if-goto b 翻译成目标语言时，应该使用完整的符号“f$b”，而不是“b”
(FunctionName)	每个 VM 函数 f 应该生成一个符号“f”，以指代函数 f 在目标计算机指令内存中的地址入口
return-address	每个 VM 函数调用应该生成并在翻译后的代码中插入唯一的标号，它代表被调用函数（called function）的返回地址，即内存单元的地址（目标计算机的内存），该地址就是紧接着调用命令之后的指令地址

图 8.6　所有 VM-on-Hack 标准映射规定的专用汇编符号

引导程序（bootstrap）代码 当 VM-Hack 翻译器编译 VM 程序（一组.vm 文件）时，会产生一个用 Hack 汇编语言编写的.asm 文件。该文件必须符合规范。标准映射规定：（1）VM 堆栈的初始地址必须被映射到 RAM[256]；（2）经过编译后的 VM 程序所执行的第一个 VM 函数必须是 Sys.init（参看 8.2.4 小节）。

如何在 VM 翻译器生成的.asm 文件中执行这个初始化呢？在第 5 章构建 Hack 计算机硬件时，我们的设计是：在重置（reset）时获取位于 ROM[0]位置的字并且执行。因此，起始于 ROM 地址 0 的代码段，称为**引导程序代码**（*bootstrap code*），是计算机“启动”时要执行的第一段代码。于是，根据前面的介绍，计算机的引导程序代码应该执行下列操作（以机器语言的形式）：

```
SP=256                  //将堆栈指针初始化为 0x0100
call Sys.init           //开始执行(翻译后的)Sys.init
```

Sys.init 将调用主程序中的主函数，然后进入无限循环。这样翻译后的 VM 程序就进入运行状态。

“程序（program）”、“主程序（main program）”和“主函数（main function）”的概念与编译过程有关，在不同的高级语言中它们的概念并不相同。例如，在 Jack 语言中，默认的是自动开始运行的第一个程序单元就是 Main 类中的 main 方法。同样，当让 JVM 去执行给定的类（比如 Foo）时，JVM 会去寻找并执行 Foo.main 方法。通过正确地编写 Sys.init ，每种语言编译器都能执行这样的“自动”启动程序。

8.3.2 范例 Example

正数 n 的阶乘可以用迭代方程 $n!=1\cdot 2\cdot \ldots \cdot (n-1)\cdot n$ 来计算。图 8.7 中给出了这个算法的实现。

现在重点分析图 8.7 中 fact 函数代码部分中的 call mult 命令。图 8.8 显示了与这个调用相关的三个堆栈状态，通过实例说明了函数调用协议。

如果忽略图 8.8 中的中间堆栈实例，会发现 fact 函数已经设置好了一些参数，并且调用 mult 应用这些参数来进行运算（左边的堆栈实例）。当 mult 返回时（右边的堆栈实例），fact 函数先前设置的参数已经被 mult 函数的返回值所代替。

```
function p
...
// 计算 4!
push constant 4
call fact 1 //1 arg
...
```

```
function fact 2 //2 个局部变量
// 返回给定参数的阶乘
   push constant 1
   pop local 0 //result=1
   push constant 1
   pop local 1 //j=1
label loop
   push constant 1
   push local 1
   add
   pop local 1 //j=j+1
   push local 1
   push argument 0
   gt
   if-goto end //if j>n goto end
   push local 0
   push local 1
   call mult 2 //压入 2 个参数
   pop local 0 //result=mult(result,j)
   goto loop
label end
   push local 0
   return
```

```
function mult 2
// (2 个局部变量)
//argument 0 乘以 argument1
//代码在图 7.9 中
//已经给出了。
...
//返回结果:
push local 0
return
```

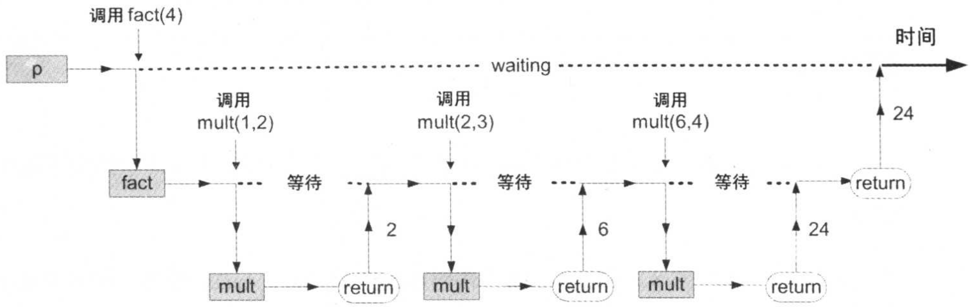


图 8.7 被函数调用的生命周期。任意函数 p 调用函数 fact，而 fact 又调用函数 mult 若干次。垂直箭头代表控制权从一个函数到另一个函数的转移。在任意时刻，只能有一个函数在运行，所有调用链中的其他函数都在等待其返回。当一个函数返回时，该函数的调用者（调用当前函数的函数）继续执行

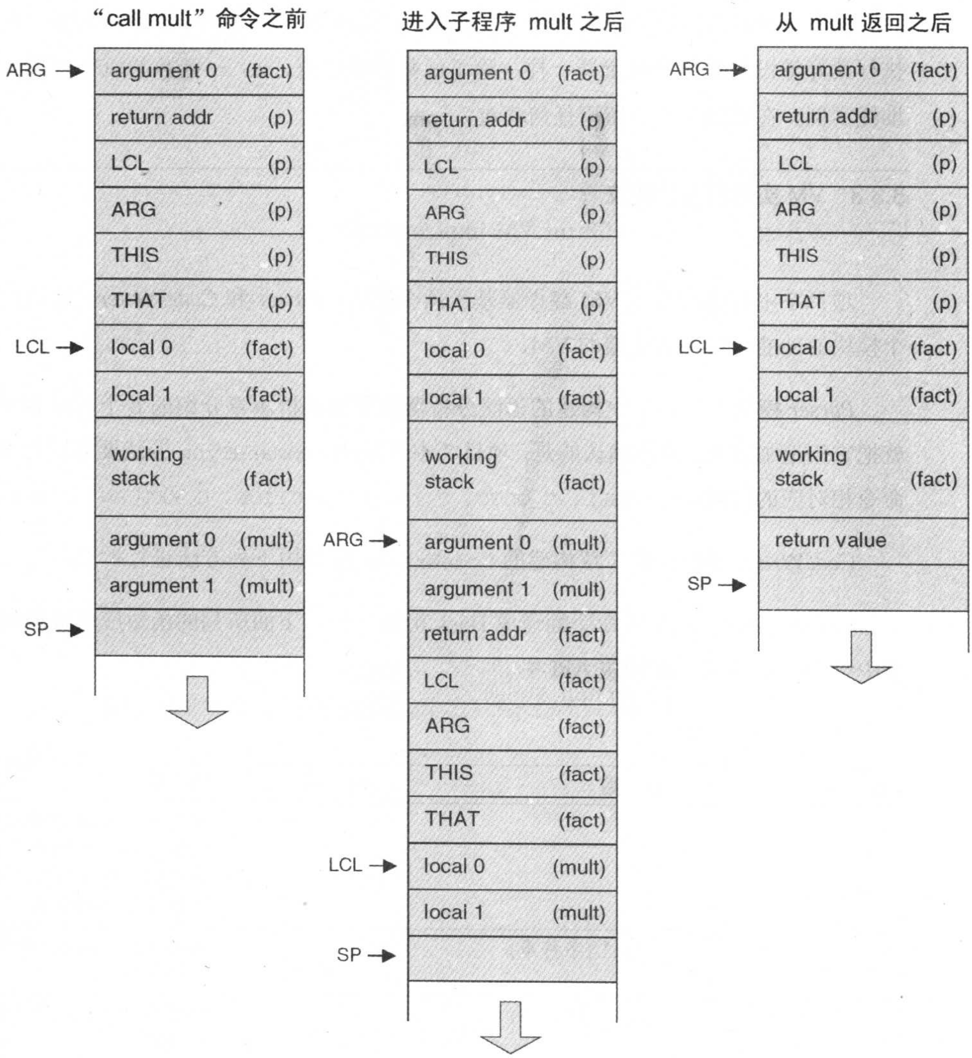


图 8.8 结合图 8.7，针对函数调用 `call mult`，给出了全局堆栈的动态描述。指针 SP、ARG 和 LCL 并非属于 VM 抽象，但 VM 实现应用它们将堆栈映射到宿主 RAM 上

换句话说，当被调用函数的任务完成之后，调用函数也得到了期望的服务，程序继续执行就好像没有发生任何事情一样：除了函数的返回值，mult 函数的执行过程（中间的堆栈实例）在堆栈上不会留下任何痕迹。

8.3.3 VM 实现的设计建议
Design Suggestions for the VM Implementation

项目 7 中构建的基本 VM 翻译器基于两个模块：*Parser* 和 *CodeWriter*。通过扩展这两个模块的功能可以实现完整的 VM。

Parser **模块** 项目 7 中构建的语法分析器还不能解析本章介绍的 6 个 VM 命令，这里就把它们追加进来。需要确认的是：项目 7 中开发的 commandType 方法要返回与 6 个 VM 命令相对应的常数：C_LABEL、C_GOTO、C_IF、C_FUNCTION、C_RETURN、C_CALL 。

CodeWriter **模块** 第 7 章构建的 CodeWriter 应该用下列方法进行扩充。

CodeWriter：将 VM 命令翻译成 Hack 汇编代码。下面所列的函数应该被添加到第 7 章中给定的 CodeWriter 模块 API 中。

程　序	参　数	返　回	功　能
writeInit	—	—	编写执行 VM 初始化的汇编代码，也称为引导程序代码。该代码必须被置于输出文件的开头
writeLabel	label (字符串)	—	编写执行 label 命令的汇编代码
writeGoto	label (字符串)	—	编写执行 goto 命令的汇编代码
writeIf	label (字符串)	—	编写执行 if-goto 命令的汇编代码
writeCall	functionName (字符串) numArgs (int)	—	编写执行 call 命令的汇编代码
writeReturn	—	—	编写执行 return 命令的汇编代码
writeFunction	functionName (字符串) numLocals (int)	—	编写执行 function 命令的汇编代码

8.4 观点
Perspective

子程序调用（subroutine calling）和程序流程控制（program flow）的概念对于所有高级语言都是很基本的。这意味着在将程序转换成二进制代码过程中的某些地方，必须考虑与其实现相关的一些复杂操作。在 Java、C#和 Jack 中，这个任务落在 VM 层上。如果 VM 是**基于堆栈**（*stack-based*）的，就能很好地解决这些复杂问题，正如在本章所见。一般而言，以“子程序调用和迭代的实现”作为基本特性的虚拟机都能表达有意义的抽象。

当然这只是对实现机制的选择。一些编译器直接处理子程序调用的细节，而根本不用 VM。其他编译器则使用各种形式的 VM，但是它们对程序调用的处理并不是必须的。最后，在一些体系结构中大多数子程序调用功能是直接被硬件处理的。

接下来的两章将开发 Jack-VM 编译器。由于该编译器的后端程序已经在第 7 章和本章中实现，所以编译器的开发就是相对容易的任务了。

8.5 项目
Project

目标 将项目7中构建的基本VM翻译器扩展成完整的VM翻译器。增加处理VM语言的程序控制流和函数调用命令的功能。

资源（与项目7相同） 要用到两个工具：用来实现VM翻译器的编程语言，以及与本书配套的CPU仿真器。该仿真器可以执行VM翻译器生成的机器代码，这是间接测试VM翻译器正确性的方法。本项目中的另一个方便的工具就是与本书配套的可视化VM仿真器。该程序允许在你开始构建自己的VM实现之前，先感受一下VM实现的运行效果。想了解更多关于这个工具的信息，读者可以参考VM仿真器指南。

约定 编写完整的VM-Hack翻译器，扩展第7章中构建的翻译器，遵守VM规范第II部分（8.2节）和Hack平台上VM标准映射（8.3.1节）。用其翻译下面提供的VM程序，生成用Hack汇编语言编写的相应程序。当这些汇编程序在提供的CPU仿真器上执行时，根据提供的测试脚本和比较文件给出运行结果。

测试程序

建议分两步来完成翻译器的实现。首先实现**程序控制流**命令，然后再实现**函数调用**命令。这样就可以应用所提供的测试程序来逐步地测试你的实现。

对于每个程序`Xxx`，`XxxVME.tst`脚本允许在VM仿真器上运行该程序，你可因此而熟悉程序的操作。在使用你实现的VM翻译器将程序编译后，可利用`Xxx.tst`和`Xxx.cmp`脚本在CPU仿真器上测试编译后的汇编代码。

程序控制流命令测试程序

- `BasicLoop`：计算 1+2+⋯+*n*，然后将结果压入堆栈。该程序测试 VM 语言的 `goto` 和 `if-goto` 命令的实现。
- `Fibonacci`：计算 Fibonacci 数列的前 *n* 个数并且将它们存入内存。这个典型的数组操纵程序提供了更具挑战性的 VM 分支命令的测试。

函数调用命令测试程序

- `SimpleFunction`：执行简单的计算并返回结果。该程序提供了 `function` 和 `return` 命令实现的基本测试。
- `FibonacciElement`：这个程序提供了 VM 函数调用命令、引导程序和其他 VM 命令实现的全面测试。

  程序路径包含两个`.vm`文件：

  - `Main.vm`包含名为`fibonacci`的函数。该迭代函数返回 Fibonacci 数列的第 *n* 个元素；
  - `Sys.vm`包含名为`init`的函数。该函数调用`Main.fibonacci`函数，其中 *n*=4，然后进入无限循环。

因为整个程序由两个`.vm`文件组成，所以整个路径必须被编译，以生成`FibonacciElement.asm`文件（单独编译每个`.vm`文件会产生两个独立的`.asm`文件，这并不是我们所需的）。

- `StaticTest`：VM 实现对静态变量处理的全面测试。包含两个`.vm`文件，每个文件代表一个单独的类文件的编译以及`Sys.vm`文件。应该对整个路径进行编译以生成`StaticsTest.asm`文件。

（前面介绍过，根据 VM 规范，VM 实现生成的引导程序代码必须包含对`Sys.init`函数的调用）。

提示

初始化 为了使任何编译后的 VM 程序能够开始运行，编译后的 VM 程序必须包含一段引导启动代码以使 VM 实现在目标平台上得以运行。另外，为了使任何 VM 代码能正确

操作，VM 实现必须保存宿主 RAM 中的指定位置的虚拟段的基地址。本项目中的前三个测试程序假设启动代码未被实现，它们包含“手动地”执行必要初始化的测试脚本。最后两个程序假设启动代码已经是 VM 实现中的一部分。

测试/调试 对于 5 个测试程序，按照下列步骤来操作：

1. 在提供的 VM 仿真器上运行程序，使用 `XxxVME.tst` 测试脚本，以此来熟悉程序的行为；

2. 使用部分实现的翻译器来编译`.vm`文件，生成一个单一的`.asm`文本文件，该文件包含用翻译后的 Hack 程序；

3. 观察翻译后的`.asm`程序。如果有明显的语法（或其他）错误，修改并调试你的翻译器；

4. 使用提供的`.tst`和`.cmp`文件在 CPU 仿真器上运行翻译后的`.asm`程序。如果存在运行期错误，修改并调试你的翻译器。

注意：本章中提供的测试程序都是精心设计的，用于对 VM 的每一步实现过程进行单元测试。因此，最好按照建议的步骤来实现 VM 翻译器，然后在每一阶段使用合适的测试程序进行测试。颠倒实现步骤会使测试程序产生错误。

工具 与项目 7 相同。

第 9 章　高级语言
High-Level language

High thoughts need a high language.
至高的思想需要至高的语言。
——Aristophanes（公元前 448 ~ 380），古希腊剧作家

到目前为止本书所介绍的硬件系统和软件系统都是底层系统，意味着人们一般不太会和它们直接打交道。本章将介绍一门高级语言，名为 Jack，程序员可以通过它来编写高级程序。Jack 是简单的基于对象（object-based）的语言。Jack 具有现代语言（如 Java 和 C#）的基本特性和风格，但是语法相对简单，并且不支持继承。尽管 Jack 很简单，它仍然是一种通用语言，能够被用来创建很多应用程序。Jack 比较适合于编写一些简单的交互式小游戏，比如 Snake（贪食蛇）、Tetris（俄罗斯方块）、Pong（碰球）——这些程序的完整 Jack 源代码包含在与本书配套的软件包里面。

Jack 的引入标志着本书旅程最后部分的开始。在第 10 章和 11 章中将编写一个将 Jack 程序翻译成 VM 代码的编译器，在第 12 章中将开发基于 Jack/Hack 平台的操作系统，利用 Jack 语言编写代码，以此来完成计算机的构建。因此有必要声明，本章的目的不是让你成为 Jack 程序员，而是为今后开发编译器和操作系统奠定基础。

如果你对任何现代面向对象编程语言有一些经验，那么对 Jack 就能很快上手。因此，**背景知识**部分一开始就会介绍一些典型的编程范例，**规范详述**部分则会对该语言的功能作完整地描述并对标准程序库进行详细介绍。**实现**部分给出了一些典型 Jack 应用的屏幕截图，并对如何在 Hack 平台上编写类似的程序提供了思路。最后的**项目**部分提供了一些关于编译和调试 Jack 程序的细节阐述。

本章的所有程序都能用与本书配套的 Jack 编译器进行编译。编译后的 VM 代码也可以在配套的 VM 仿真器上运行。读者也可以使用在第 7 章至第 8 章中的 VM 翻译器和第 6 章中的汇编编译器将 VM 代码进一步翻译成二进制代码，由此生成的机器代码可以在第 1 至 5 章中构建的硬件平台上运行。

在这里有必要重申一下，Jack 是相对比较乏味、简单的语言。然而，简单有简单的道理。首先，读者可以很快地学习（和忘记）Jack——不超过 1 个小时。其次，Jack 语言易于被应用在一些简单的编译技术中。这样，我们可以很简单地编写很好的 **Jack 编译器**（这正是在第 10 章和 11 章中要做的）。也就是说，Jack 的简单结构能帮助我们揭示一些隐含在现代语言（比如 Java 和 C#）之下的软件工程原理。这样就可以亲自动手构建 Jack 编译器和运行期环境，而不是去想当然。好了，下面开始了解 Jack 吧！

9.1 背景知识 Background

Jack 语言结构简单，不言自明。因此，把语言规范放在下一节中介绍，先来看一些范例。首先会引入经典的 *Hello World* 程序。第二个例子将会介绍过程化编程（procedural programming）模型和数组处理。第三个例子介绍如何将基本语言扩展为抽象的数据类型。最后一个例子介绍应用该语言的对象处理能力来实现链表。

9.1.1 范例 1：Hello World Example 1: Hello World

当我们让 Jack 运行期环境去执行给定的程序时，执行总是从 `Main.main` 函数开始的。因此，每个 Jack 程序必须至少包含一个名为 `Main` 的类，而且这个类必须包含一个名为 `Main.main` 的函数。该语法规则如图 9.1 所示。

Jack 语言内置了**标准程序库**（*standard library*），第 9.2.7 节给出了其完整 API。这个库通过不同的抽象和服务（比如数组、字符串、数学函数、内存管理、输入/输出函数等）来扩展基本语言。图 9.1 中的程序中调用了两个这样的函数，导致屏幕打印“`Hello World`”的程序运行结果。

```
/** Hello World 程序 */
class Main {
  function void main() {
    /*使用标准库函数来打印文本*/
    do Output.printString("Hello World");
    do Output.println(); //换新行
    return;
  }
}
```

图 9.1 Hello World

9.1.2 范例 2：过程化编程和数组处理

Example 2: Procedural Programming and Array Handling

Jack 支持过程化编程（procedural programming）中典型的语言结构。Jack 也包括用于声明和处理数组的基本命令。图 9.2 中的程序描述了这些特性，它处理数组的输入和输出并对该数组进行平均值的计算。

Jack 程序使用内置的 `Array` 类来声明和构建数组，该类是 Jack 标准库的一部分。注意到 Jack 数组是没有指定类型的，它可以包含任意类型：整数、对象，等等。

9.1.3 范例 3：抽象数据类型

Example 3: Abstract Data Types

每种编程语言都有一组固定的基本数据类型，Jack 支持三种基本数据类型：`int`、`char` 和 `boolean` 。程序员可以通过创建新的抽象数据类型对基本数据类型进行扩展。比如，假设希望赋予 Jack 处理**有理数**的能力，即处理形如 *n/m*（*n* 和 *m* 为整数）的对象。可以创建单独的类来为 Jack 程序提供“分数”概念的抽象。我们就称这个类为 `Fraction` 。

定义类的接口 合理的方法是，指定该分数抽象中的属性和服务。图 9.3a 给出了这样的**应用程序接口**（API，*Application Program Interface*）。

```
/** 计算整数序列的平均值 */

class Main {
  function void main() {
    var Array a;
    var int length;
    var int i, sum;

    let length = Keyboard.readInt("How many numbers? ");
    let a = Array.new(length); //构造数组
    let i = 0;

    while (i < length) {
      let a[i] = Keyboard.readInt("Enter the next number: ");
      let sum = sum + a[i];
      let i = i + 1;
    }

    do Output.printString("The average is: ");
    do Output.printInt(sum / length);
    do Output.println();
    return;
  }
}
```

图 9.2 过程编程和数组处理

Jack 语言中，在当前对象上（用 this 来指代）的操作用**方法**（*methods*）来表示，而在类级别上的操作（等同于 Java 中的静态方法）用**函数**（*functions*）来表示。创建新对象的操作称为**构造函数**（*constructor*）。

类的使用 API 对于不同的人有不同的含义。如果你是负责**实现**分数类(fraction class）的程序员，就可以把它的 API 看作是必须通过某种方式来实现的约定。而如果你是需要**使用**该分数类的程序员，就可以把 API 看作是分数类服务提供者的文档，以此 API 来创建分数对象并提供分数相关的操作。现在来考虑后一种情况，参看图 9.3b 中列出的 Jack 代码。

图 9.3b 说明了重要的软件工程原理：对于任何给定用户的抽象，用户不需要知道其内在**实现**细节。

```
// Fraction是表示分数n/m的对象，这里n和m都是整数
field int numerator, denominator             //Fraction对象属性
constructor Fraction new(int a, int b)       //返回新的Fraction对象
method int getNumerator()                    //返回该分数的分子
method int getDenominator()                  //返回该分数的分母
method Fraction plus(Fraction other)         //以分数形式返回该分数和另一个分数的和
method void print()                          //以“分子/分母”的格式打印该分数
//与分数相关的其他服务根据需要在这里指定
```

图 9.3a Fraction 类 API

```
// 计算2/3和1/5的和
class Main {
  function void main() {
    var Fraction a, b, c;
    let a = Fraction.new(2,3);
    let b = Fraction.new(1,5);
    let c = a.plus(b); // 计算c = a + b
    do c.print(); //应该打印文本"13/15"
    return;
  }
}
```

图 9.3b 使用 Fraction 抽象

```
/**提供Fraction类型和相关的服务*/
class Fraction {
  field int numerator, denominator;
  /**利用给定的分子和分母构造(经过化简的)分数*/
  constructor Fraction new(int a, int b) {
    let numerator = a; let denominator = b;
    do reduce(); // 如果a/b没有化简，就进行化简
    return this;
  }

  /**化简该分数*/
  method void reduce() {
    var int g;
    let g = Fraction.gcd(numerator, denominator);
    if (g > 1) {
      let numerator = numerator / g;
      let denominator = denominator / g; }
    return;
  }

  /**计算a和b的最大公约数 */
  function int gcd(int a, int b){
    var int r;
    while (~(b = 0)) { //应用Eculid算法
      let r = a - (b * (a / b)); // r = a/b的余数
      let a = b; let b = r; }
    return a;
  }

  /** 访问函数(Accessor) */
  method int getNumerator() { return numerator; }
  method int getDenominator() { return denominator; }
```

图 9.3c 一个可能的 Fraction 类的实现形式

```
  /**返回该分数和另一个分数的和
  method Fraction plus(Fraction other){
    var int sum;
    let sum = (numerator * other.getDenominator()) +
              (other.getNumerator() * denominator());
    return Fraction.new(sum, denominator *
                        other.getDenominator());
  }

  //更多与分数相关的方法: minus, times, div, 等等

  /**打印该分数*/
  method void print() {
    do Output.printInt(numerator);
    do Output.printString("/");
    do Output.printInt(denominator);
    return;
  }
} // Fraction类
```

续图 9.3c 一个可能的 Fraction 类的实现形式

这些用户只能访问该抽象的**接口**（*interface*）或称 API，并将其看作提供“与抽象相关的操作”的黑匣子

类的实现 下面转到另一个角色中去，作为真正去实现分数抽象的程序员。图 9.3c 给出了一种可行的 Jack 实现。

图 9.3c 展示了典型的 Jack 程序结构：**类**（*class*）、**方法**（*methods*）、**构造函数**（*constructors*）和**函数**（*functions*）。图 9.3c 还展示了 Jack 中支持的所有的语句类型：`let`、`do`、`if`、`while` 和 `return` 。

9.1.4 范例 4：链表实现

Example 4: Linked List Implementation

链表（*linked list*，或简称**列表**，*list*）是一种对象链，每个对象包含数据域和指向链表中另一个对象的指针。图 9.4 显示了一种链表抽象的 Jack 类实现。本例的目的是说明 Jack 语言中对象的典型处理方式。

```
/** List类提供的链表抽象*/
class List {
 field int data;
 field List next;

 /*创建新的List对象*/
 constructor List new(int car, List cdr) {
  let data = car;
  let next = cdr;
  return this;
 }

 /* 从尾部开始递归释放整个链表 */
 method void dispose() {
  if (~(next = null)) {
   do next.dispose();
  }
  // 应用操作系统提供的服务来回收该对象所占用的内存
  do Memory.deAlloc(this);
  return;
 }

 // 更多和链表相关的方法
} // List类
```

```
/*创建用来保存数字 (2, 3, 5) 的链表。(这段代码可以出现在任何类中) */
function void create235() {
 var List v;
 let v = List.new(5,null);
 let v = List.new(2,List.new(3,v));
 ... //执行某些链表操作
 do v.dispose();
 return;
}
```

图 9.4 在链表中的对象处理

9.2 Jack 语言规范详述
The Jack Language Specification

现在来对 Jack 语言作正式完整的描述，包括语法要素、程序结构、变量、表达式和语句构成。必要时该语言规范可被看作是技术参考。

9.2.1 语法要素
Syntactic Elements

Jack 程序是一系列用任意数量空格和注释而分隔开来的**字元**（*tokens*）。这些字元可以是**符号**（*symbol*），**保留字**（*reserved words*），**常数**（*constants*）和**标识符**（*identifiers*），如图 9.5 所示。

9.2.2 程序结构
Program Structure

Jack 中的基本编程单元是**类**（*class*）。每个类存在于独立的文件中，可独立编译。类的定义具有如下的格式：

```
class 名称 {
   成员字段（field）[1]和静态变量声明          //必须在子程序声明之前。
   子程序声明                                  //构造函数声明、方法声明，以及函数声明。
}
```

每个类声明先指定类的名称，通过该名称，对应的类可以在全局范围内被访问到。接下来是一组零个或多个成员**字段**（*field*）和**静态变量**（*static variable*）声明。最后是一组一个或多个**子程序**（*subroutine*）声明，每个声明定义**方法**、**函数**或**构造函数**。方法“属于”对象并且提供对象的功能；函数一般“属于”类，而且不与某个特定的对象相关（跟 Java 中的**静态方法类似**）；构造函数“属于”类，在被调用时生成该类的对象实例。

所有的子程序的声明都具有以下格式：

```
subroutine 类型 名称 (参数列表) {
  局部变量声明
  语句
}
```

[1] 成员字段（field）是指类中相对独立的程序单元。——审校者

空格和注释	忽略空格符，换行符和注释。 支持的注释格式如下： // 到行尾的注释 /* 普通注释 */ /** 用于 API 文档的注释*/	
符号	( )	用于封装算术表达式和参数列表
	[ ]	用于数组索引
	{ }	用于封装程序单元和语句
	,	变量列表分隔符
	;	语句终结符
	=	赋值和比较运算符
	.	类成员
	+ - * / &	~ < >	运算符
保留字	class，constructor，method，function	程序组件
	int，boolean，char，void	原始的基本类型
	var，static，field	变量声明
	let，do，if，else，while，return	表达式
	true，false，null	常数
	this	对象引用
常数	*Integer* 常数必须是正数，并且是标准的十进制格式，比如 1984 。负整数，如 -13 不是常数，而是对整常数取负值的一元运算表达式 *String* 常数用两个双引号字符括起来，可以处理除换行符和双引号（这两种字符由标准库中的函数 String.newLine()和 String.doubleQuote()来提供）之外的任何字符 *Boolean* 常数可以是 true 或 false 常数 null 表示一个空引用	
标识符	标识符由任意长度的字符串组成，这些字符可以是字母（A~Z，a~z），数字（0-9），和 “_”。但第一个字符必须是一个字母或者 “_” 语言是区分大小写的。因此 x 和 X 被认为是不同的标识符	

图 9.5 Jack 语法要素

其中，子程序可以是构造函数、方法或函数。每个子程序都有供调用的**名称**，以及描述该子程序返回值的**类型**。如果子程序不返回任何值，则返回类型为 void；如果子程序有返回值，则返回类型可以是该语言所支持的任何基本数据类型，或者标准库提供的“类”类型（class type），或者应用中其他类提供的“类”类型。构造函数可以取任意名称，返回值必须返回该类的对象。因此，构造函数的返回类型必然是它所属的类。

跟 Java 一样，**Jack 程序**也是由一个或多个类构成。必须有一个类被命名为 Main，该类至少必须包括名为 main 的函数。当执行某一路径下的 Jack 程序时，Jack 运行期环境将自动地开始执行 Main.main 函数。

9.2.3 变量
Variables

Jack 语言中的变量在使用之前必须先被显式地声明出来。在 Jack 中有四种类型的变量：**成员字段**（*field*），**静态变量**（*static variable*），**局部变量**（*local variable*）和**参数变量**（*parameter*），每种变量都有其适用范围。

数据类型 变量可以是一种**基本**数据类型（int/char/boolean），或是**对象**类型（*object type*），即类名称。实现这种数据类型的类可以是 Jack 标准库中的类（比如 String 和 Array），也可以是在程序路径下的其他类。

基本类型 Jack 中有三种基本数据类型：

- int ：16-位 2 补码
- boolean ：假（*false*）和真（*true*）
- char ：Unicode 字符

当基本类型变量被声明时，就会在内存中为其分配对应的内存单元。比如，有如下声明：var int age; var boolean gender; 那么编译器会去创建变量 age 和 gender 并且为它们分配对应的内存单元。

对象类型（*object types*） 每个类定义了对象类型。跟 Java 一样，对象的声明实际只是创建一个指向该对象的引用变量（指针[2]），真正用于存储该对象的内存单元只有在调用构造函数来构造对象时才会被分配。图 9.6 给出了例子。

[2]这里所说的指针是抽象概念，并非特指某些高级语言中实现的具体的指针技术。——审校者

```
// 这段代码假设已经实现了Car类和Employee类
// Car对象有model和licensePlate两个成员
// Employee对象有name和Car两个成员
var Employee e, f;       // 创建包含空引用的变量e, f
var Car c;               // 创建包含空引用的变量c
...
let c = Car.new("Jaguar","007")     //构造新的Car对象
let e = Employee.new("Bond",c)      //构造新的Employee对象
// 这时，c和e分别指向Car 和Employee这两个对象在内存中所占有内存段的基地址
let f = e; //只是拷贝了对象的引用，并没有构造新对象
```

图 9.6 对象类型（范例）

Jack 标准程序库提供了两个内置对象类型（类）：`Array` 和 `String` 。

数组（*Array*） 数组使用内置类 `Array` 来声明。数组是一维的，第一个编号总是 0（多维数组可以通过数组中嵌套数组来实现）。数组中的数据项没有特定的类型，同一数组中的不同数据项可以是不同的类型。数组的声明仅仅创建了引用（reference），数组的实际创建工作是通过调用 `Array.new(length)` 构造函数来完成的。对数组元素的访问使用 `a[j]` 符号。图 9.2 给出了使用数组的例子。

字符串（*Strings*） 字符串使用内置类 `String` 来声明。Jack 编译器能够认出 `"xxx"` 并且将其当作是某个 `String` 对象的内容。我们可以通过使用 `String` 的方法来访问和修改 `String` 对象。比如：

```
var String s;
var char c;
...
let s = "Hello World";
let c = s.charAt(6);   // "W"
```

类型转换 Jack 是弱类型（weakly typed）语言。语言规范并没有定义从一种类型转为另一种类型的结果，允许或禁止某种转换是根据不同的编译器而定的（放开这个自由度是有意而为之，这样就能忽略类型问题，以便构建最小规模的 Jack 编译器）。

因此，所有的 Jack 编译器都期望能允许并且自动执行下面的任务：

- 字符和整数能够根据 Unicode 规范在必要时相互转换。比如：

```
var char c; var String s;
let c = 33;  // 'A'
// 同样地：
let s = "A"; let c=s.charAt(0);
```

- 整数可被赋给任何对象类型的引用变量（reference variable），这时该整数被当作是内存中的地址。比如：

```
var Array a;
let a = 5000;
let a[100] = 77;  // 内存地址 5100 的值被置为 77
```

- 对象变量（其类型为某个类）可被转换成 `Array` 变量，反之亦然。经过转换就可以像访问数组中的数据项一样去访问对象中的成员，反之亦然。比如：

```
// 假设 Complex 类有两个 int 字段：re 和 im。
var Complex c; var Array a;
let a = Array.new(2);
let a[0] = 7; let a[1] = 8;
let c = a; // c==Complex(7,8)
```

变量的类型和作用域 Jack 有四种类型的变量。**静态变量**（*static variable*）定义在类这一级（class level），被该类的所有对象共享。比如，`BankAccount` 类可以定义 `totalBalance` 静态变量来保存所有银行账户的金额总数，其中每个账户都是 `BankAccount` 类的实例对象。

变量类型	定义/描述	声明的位置	作用域
静态变量	**static** *type name1, name2, ...;* 每个静态变量只存在唯一副本，并被该类的所有对象实例共享（就象 Java 中的*私有静态变量*一样）	在类中声明	在声明它的类中有效
字段变量	**field** *type name1, name2, ...;* 类的每个对象实例都各自拥有一个成员变量的私有副本（就象 Java 中的*私有对象变量*一样）	在类中声明	在声明它的类中有效，函数除外
局部变量	**var** *type name1, name2, ...;* 当子程序被调用时，局部变被分配在堆栈里，当子程序返回时，局部变量被释放。	在子程序中声明	其被定义的子程序中有效
参数变量	*type name1, name2, ...;* 用于指定子程序的输入，比如： `function void drive (Car c, int miles)`	出现在子程序声明的参数列表中	在其被定义的子程序中有效

图 9.7　Jack 语言中的变量类型（表中，子程序可以是函数、方法或构造函数）

成员字段变量（*field variable*）用于定义类对象的属性（properties），比如，`BankAccount` 类的成员变量 `owner` 和 `balance` 。**局部变量**（*local variable*），被子程序使用，仅仅存在于子程序的生存周期内，**参数变量**（*parameter variable*）用于传递变量给子程序。比如，`BankAccount` 类可以包括方法 `method void transfer(BankAccount from, int sum)`，其中定义了两个参数 `from` 和 `sum`。如果 `joeAccount` 和 `janeAccount` 是两个 `BankAccount` 类型的变量，那么命令 `joeAccount.transfer(janeAccount,100)` 的作用就是从 Jane 的账户中转账 100 元到 Joe 的账户。

图 9.7 给出了 Jack 语言支持的所有变量类型的正式描述。变量的**作用域**（*scope*）是程序中该变量可以被识别并发挥作用的范围。

语　句	语　法	描　述
let	**let** 变量 = 表达式; 或者 **let** 变量[表达式] =表达式;	赋值操作（变量可以是一个单一值或一个数组）。变量类型可以是静态、局部、成员或参数类型
if	**if** (表达式) { 　语句 } **else** { 　语句 }	典型的 *if-else* 语句。规定必须使用花括号，即使花括号中只有一条语句
while	**while** (表达式) { 　语句 }	典型的 *while* 语句。规定必须使用花括号，即使花括号中只有一条语句
do	**do** 函数或方法调用;	用于调用函数或方法，忽略返回值
return	**Return** 表达式; 或者 **return**;	用于返回子程序的值。函数和返回 void 值的方法必须采用第二种形式。构造器必须返回 this

图 9.8　Jack 语句

9.2.4 语句
Statements

Jack 语言支持五种常见的语句形式。图 9.8 给出了它们的定义和描述。

9.2.5 表达式
Expressions

根据 Jack 语言的规则，图 9.9 给出了 Jack 表达式的定义。

Jack **表达式**必须是下列之一：

- 常数；
- 在作用域内的变量名（变量可以是静态、局部、成员或参数类型）；
- 关键字 `this`，引用当前对象（不能用于函数中）；
- 数组语法是：*数组名称[表达式]*，其中*数组名称*是 `Array` 类型的变量名；
- 返回值为非空（non-void）类型的*子程序调用*（subroutine call）；
- 一元运算符“-”或“~”作前缀的表达式：
  - -*表达式*：算术求反；
  - ~*表达式*：布尔求反（对于整数，则按位取反）；
- 形如“*表达式 运算符 表达式*”的表达式，其中运算符（operator）是以下二元运算符中的一种：
  - `+ - * /`　整数算术运算符；
  - `& |`　布尔“与”和布尔“或”（对于整数，则按位操作）运算符；
  - `< > =`　比较运算符；
- (*表达式*)：位于圆括号内的表达式。

图 9.9　Jack 表达式

运算符优先级和计算顺序　除了规定括号中的表达式要先计算外，Jack 语言中没有定义运算符的优先级。比如表达式 `2+3*4` 的结果就可能是 20 或者 14，只有这个 `2+(3*4)` 表达式的结果才确定是 14。在此种表达式中使用圆括号，这为 Jack 编程带来一点麻烦。然而，缺少运算符优先级顺序的规定，是基于一定的考虑，因为这样做可以简化 Jack 编译器的编写。当然，对于不同的语言编译器实现，可以在必要时规定运算符的优先级并将其添加到语言规范文档中去。

9.2.6　子程序调用
Subroutine Calls

子程序调用唤起方法（method）、函数（function）和构造函数（constructor），使用这样的语法形式：*methodName(argumentlist)*（即“*子程序名(参数列表)*”）。参数的数量和类型必须与子程序的声明中所定义的相匹配。即使参数列表为空，圆括号也是必须要有的。

```
class Foo {
  // 这里省略一些子程序声明。
  ...
 Method void f() {
   var Bar b;               // 声明Bar类型的局部变量
   var int i;               // 声明基本数据类型int的局部变量
   ...
   do g(5,7)                // （对当前对象实例）调用Foo类的g方法
   do Foo.p(2)              // 调用Foo类的p函数
   do Bar.h(3)              // 调用Bar类的h函数
   let b = Bar.r(4);        // 调用Bar类的构造函数或r函数
   do b.q()                 // （对当前对象实例）调用Bar类的q方法
   Let i = w(b.s(3), Foo.t())    // 对当前对象实例调用w方法，
                                 // 调用b对象的s方法，
                                 // 调用Foo类的函或或构造函数t
   ...
 }
}
```

图 9.10 子程序调用范例

每个参数可以是具有无限复杂度的表达式。比如，`Math` 类是 Jack 标准库中的类，包含定义为 `function int sqrt(int n)`的均方根函数。那么我们可以使用 `Math.sqrt(17)`或者 `Math.sqrt((a*Math.sqrt(c-17)+3)`等等形式来调用该函数。

在类的内部，使用语法 *methodName(segment-list)*（即“*方法名(参数列表)*”）来调用方法（method），对函数和构造函数的调用必须使用全名，也就是 *className.subroutineName(argument-list)*（即“*类名.子程序名(参数列表)* ”）。在类的外部进行调用时，对类的函数和构造函数的调用也必须使用全名，而对于方法则必须使用语法：*varName.methodName(argument-list)*（即“*变量名.方法名(参数列表)* ”），这里 *varName* 应该是已定义的对象变量。图 9.10 给出了一些例子。

对象构造（Construction）和清除（Disposal）[3] 对象的构造可以分为两个阶段。在程序定义某个对象类型的变量之后，仅仅创建了引用（指针）变量并为其分配了内存。为了完成对象的构造，程序必须调用该类的构造函数。因此，实现数据类型的类（比如图 9.3c 中的 `Fraction`）至少必须包含一个构造函数。构造函数可取任意名称，但习惯上使用 `new`

[3]这里所说的“清除（disposal）”类似于 C++语言里面的“析构（distruction）”以及 Java 语言里面的“清除（dispose）”。后续提到的所谓“`dispose()`”函数，与 C++语言中的“析构函数（distructor）”以及 Java 语言中的“`dispose()`”函数类似。——审校者

来称呼它。构造函数的调用就像其他任何类的函数一样，具有以下格式：

let *varName = className.constructorName(parameter-list);*
let *变量名 = 类名.构造器名(参数列表);*

比如，`let c = Circle.new(x,y,50)`这里 x、y 和 50 是圆心在屏幕上的位置和圆的半径。调用构造函数时，编译器会要求操作系统分配足够的内存空间（实际就是一块内存段）来存放新对象。操作系统会返回该内存段的基地址，最后编译器将该地址赋予 `this` 指针（在 `Circle` 例子中，`this` 的值被赋给 `c`）。接下来，对象将被初始化，具体由构造函数中的 Jack 语句来完成。

当程序中不再需要某个对象时，该对象就可以被清除。特别需要指出的是，对象可以从内存中去配空间，可以通过使用标准库中的 `Memory.deAlloc(object)`函数收回原先分配给对象的内存空间。按照惯例，每个类都要包含 `dispose()`方法，用来封装去配操作。具体例子如图 9.4 所示。

9.2.7 Jack 标准库 The Jack Standard Library

Jack 语言本身提供了一组内置类，扩展了语言能力。该标准库也可被看作是基本的操作系统，必须提供给 Jack 语言实现。标准库包括下面的类：

- *Math* 提供基本的数学运算；
- *String* 实现字符串 String 类型和字符串相关操作；
- *Array* 实现数组 Array 类型和数组相关操作；
- *Output* 处理屏幕上的文本输出；
- *Screen* 处理屏幕上的图像输出；
- *Keyboard* 处理键盘的用户输入；
- *Memory* 处理内存操作；
- *Sys* 提供与程序执行相关的服务。

Math 该类实现各种数学运算操作：

- function void **init**()：仅供内部使用；
- function int **abs** (int x)：返回 *x* 的绝对值；

- function int **multiply** (int x, int y)：返回 *x* 与 *y* 的乘积；
- function int **divide** (int x, int y)：返回 *x*/*y* 除法结果的整数部分；
- function int **min** (int x, int y)：返回 *x* 和 *y* 中的较小值；
- function int **max** (int x, int y)：返回 *x* 和 *y* 中的较大值；
- function int **sqrt** (int x)：返回 *x* 的平方根的整数部分。

String 该类实现 String 数据类型以及与字符串相关的操作：

- constructor String **new** (int maxLength)：构建新的空字符串（长度为 0），最多能包含 maxLength 个字符；
- method void **dispose** ( )：清除字符串；
- method int **length** ( )：返回字符串的长度；
- method char **charAt** (int j)：返回字符串第 *j* 个位置上的字符；
- method void **setCharAt** (int j, char c)：将字符串中第 *j* 个元素置为字符 *c*；
- method String **appendChar** (char c)：在字符串末尾追加字符 *c* 并返回整个字符串；
- method void **eraseLastChar** ( )：删除字符串中最后一个字符；
- method int **intValue** ( )：返回字符串（或是“从最左边开始直到遇到非数字字符为止”的子串）的整数值；
- method void **setInt** (int j)：以字符串形式保存 *j* 所代表的整数；
- function char **backspace** ( )：返回退格字符（backspace）；
- function char **doubleQuote** ( )：返回双引号（"）字符；
- function char **newLine** ( )：返回换行符。

Array 该类构造和清除数组

- function Array **new** (int size)：构造大小为 size 的新数组；
- method void **dispose** ( )：清除数组。

Output 该类提供在屏幕上打印文本的服务：

- function void **init** ( )：仅供内部使用；

- function void **moveCursor** (int i, int j)：将光标移动到第 *i* 行的第 *j* 列，并且删除此位置上的字符；
- function void **printChar** (char c)：在光标处打印 *c*，并在该行将光标向前移动一格；
- function void **printString** (String s)：在光标处开始打印字符串 *s*，并将光标向前移动到打印结束的位置；
- function void **printInt** (int i)：在光标处开始打印整数 *i*，并且将光标向前移动到打印结束的位置；
- function void **println** ( )：将光标移动到下一行的起始处；
- function void **backspace** ( )：在该行将光标向后移动一格。

Screen　该类提供在屏幕上绘制图形的服务。列索引从 0 开始，从左至右递增。行索引从 0 开始，自上而下递增。屏幕的尺寸大小依赖于硬件（Hack 平台上的屏幕尺寸为 256 行×512 列）：

- function void **init** ( )：仅供内部使用；
- function void **clearScreen** ( )：清屏；
- function void **setColor** (boolean b)：为后续 drawXXX 命令设置绘图颜色（white = false，black=true）；
- function void **drawPixel** (int x, int y)：在坐标(x, y)处绘制像素；
- function void **drawLine** (int x1, int y1, int x2, int y2)：在像素点(x1, y1)与像素点(x2, y2)之间画一条直线；
- function void **drawRectangle** (int x1, int y1, int x2, int y2)：绘制有填充色的长方形，长方形的左上角坐标是(x1, y1)，右下角坐标是(x2, y2)；
- function void **drawCircle** (int x, int y, int r)：绘制圆心坐标为(x, y)，半径为 r（r<=181）的有填充色的圆。

Keyboard　该类提供从标准键盘上读取输入的服务：

- function void **init** ( )：仅供内部使用；
- function char **keyPressed** ( )：返回当前键盘上被按下的键所对应的字符；如果当前没有键被按下则返回 0；
- function char **readChar** ( )：等待键盘上某个键被按下后又被释放，将该键的字符返

回并显示到屏幕上；

- function String **readLine** (String message)：从键盘输入读取一行字符串（直到遇见换行字符为止），在屏幕上显示该行字符串，然后返回该字符串。该函数能处理退格（back-space）；
- function int **readInt** (String message)：从键盘输入读取一行字符串（直到遇见换行字符为止），在屏幕上显示该行字符串，然后返回其对应的整数值（直到遇见第一个非数字字符为止）。该函数也能处理退格。

Memory 该类允许直接访问宿主平台的主内存的服务：

- function void **init** ()：仅供内部使用；
- function int **peek** (int address)：返回地址为 address 的内存单元中的内容；
- function void **poke** (int address, int value)：将整数值 value 存入地址为 address 的内存单元中；
- function Array **alloc** (int size)：从内存堆（heap）中寻找并分配一块大小为 size 的内存区，返回其基地址的指针；
- function void **deAlloc** (Array o)：收回之前分配给对象的内存空间。

Sys 该类提供与程序执行相关的服务：

- function void **init** ()：调用其他 OS 类的 init 函数，进而调用 `Main.main()` 函数。仅供内部使用；
- function void **halt** ()：中止程序执行；
- function void **error** (int errorCode)：在屏幕上打印错误代码，并中止程序执行；
- function void **wait** (int duration)：等待大约 duration 毫秒后返回。

9.3 编写 Jack 应用程序 Writing Jack Applications

Jack 是一种通用的编程语言，它能在不同的硬件平台上实现。在接下来的两章将开发能生成二进制 Hack 代码的 Jack 编译器，这里有必要先来讨论 Hack 平台下的 Jack 应用程序。这一节主要介绍三个应用程序，并提供在 Jack-Hack 平台上开发应用程序的基本思路。

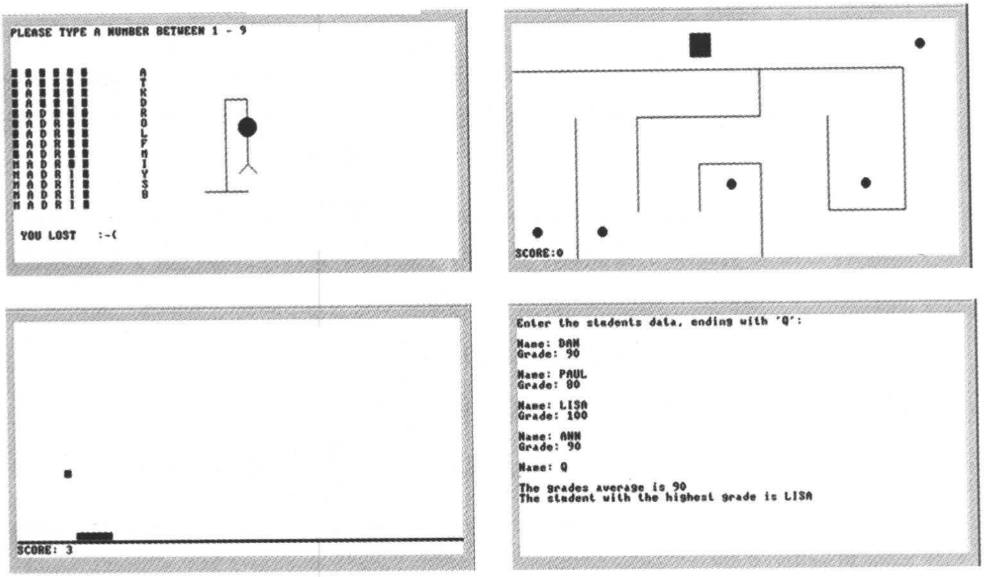


图 9.11 Jack 应用范例（运行在 Hack 计算机上）的屏幕截图。Hangman、Maze、Pong 和一个简单的数据处理程序

范例 图 9.11 中给出了 4 个应用范例。本书提供了碰球（Pong）游戏的 Jack 代码，作为在 Hack 平台上进行 Jack 编程的优秀范例。Pong 的代码是充实且有内容的，它由若干个类和几百条语句构成。该程序执行一些复杂的数学计算以计算球在屏幕上移动的方向，还要在屏幕上以动画形式描述图形对象的位移，这需要大量地使用绘图函数。为了能快速地执行上述任务，程序必须高效，意味着它必须尽可能少地进行实时计算和绘图操作。

应用程序的设计和实现 在 Hack 这样的硬件平台上开发 Jack 应用程序需要进行仔细规划。首先，应用程序设计者必须考虑硬件的物理限制，然后进行相应的计划。比如，计算机屏幕的范围限制了程序能够处理的图形化图像的尺寸。同样地，为了斟酌具体能做什么和不能做什么，设计者必须考虑语言的输入/输出命令的适用范围和平台的执行速度。

通常，在设计流程中，首先要对应用程序的行为作概念性描述。在图形和交互式程序的情况下，该描述通常表现为屏幕画面的手绘图例。在一些简单应用中，设计者可以从过程化编程来开始逐步实现。在更复杂的应用中，建议首先为应用程序创建基于对象的设计方案。这让我们能够辨识出**类**、**成员字段**和**子程序**，进而得以创建一些 API（如图 9.3a 所示）。

接下来，设计者可以用 Jack 语言来实现设计，使用 Jack 编译器来编译类文件（class file）。编译器生成的测试代码和调试代码依赖于目标平台。在 Hack 平台中，测试和调试工作通常都是通过使用 VM 仿真器来完成的。另外，设计者也可以将 Jack 程序翻译成二进制代码然后在 Hack 硬件上直接运行，或者在与本书配套的 CPU 仿真器上运行。

Jack OS Jack 程序会大量使用语言自身的标准库（也称为 Jack OS）提供的各种抽象和服务。OS 本身也是用 Jack 语言编写的，因此其可执行版本也是一组编译后的 `.vm` 文件，与编译之后的用户程序是类似的。因此，在运行任何 Jack 程序之前，首先必须将包含 Jack OS 的 `.vm` 文件拷贝到程序路径下。整个命令链如下：计算机首先运行 `Sys.init`，该函数会自动运行 `Main.main` 函数，接着 `Main.main` 函数将从用户程序和 OS 中调用各种子程序和服务。

虽然 Jack 语言的标准库可以扩展，但是读者可能希望在其他地方来锻炼提高编程能力。毕竟，我们并不期望 Jack 超出本身之外而成为你生活的一部分。因此，最好将 Jack/Hack 平台看作是给定的环境，只要充分发掘其可用潜力就行了。为受限环境下运行的嵌入式设备以及专用处理器编写软件时，程序员就是要像这样极尽硬件所能。专业人员并不把目标平台上的这些限制看成什么问题，相反，他们认为这正是展示才智的好机会。这就是为什么一些最优秀的程序员首先是在最原始的计算机上接受实践磨练。

9.4 观点 Perspective

Jack 是一门“基于对象（object-based）”的语言，这意味着它支持对象（objects）和类（classes），但是并不支持继承（inheritance）。在这方面定位介于过程化语言（比如 Pascal

或C）和面向对象语言（比如Java或C++）之间。当然Jack肯定比这些工业级强度的编程语言要“拙劣”。然而Jack的基本语法和语义与现代语言的语法和语义相差并不大。

Jack语言的一些特性还有待改进。比如，其基本类型系统是相当简单的。而且，Jack是一种弱类型（weakly typed）语言，即并没有严格地规定在定义和操作中的数据类型的一致性。读者还会感到疑惑，为什么Jack语法包含像`do`和`let`这样的关键字，为什么必须使用大括号（即使其中只包含一条简单的语句），为什么不规定运算优先级。

所有这些与常规编程语言的背离的特性，之所以被应用到Jack语言中只有一个简单的目的，那就是为了开发良好且简单的Jack编译器（正如我们接下来的两章要做的）。比如，在分析一条语句（用任何语言编写）时，如果语句的第一个字元（token）就能够说明遇到的是哪种语句，那么处理代码就要容易得多。这就是为什么Jack语法中会包含像`do`和`let`等等这样的关键字。因此，虽然Jack的这种简单性可能会给编写**Jack应用程序**带来麻烦，然而当你在下面两章中尝试编写**Jack编译器**时，会深刻感受到这种简单性带来的便利。

大多数现代语言都配有**标准库**，Jack也是如此。与Java和C#中的标准库一样，Jack的标准库也可以被看作是简单方便的操作系统接口。在Jack/Hack平台中，该OS所提供的服务是极度有限的，它不具备并发机制，因此不支持多线程或并行处理；它没有文件系统，因此不支持永久存储；它也没有通信机制。然而Jack OS也提供了一些经典的OS服务，比如以非常基本的形式提供了图形化I/O和文本I/O，标准的字符串实现，标准的内存分配和去配。此外，Jack OS还实现了各种数学函数，包括乘除法（这通常是在硬件中实现的）。我们将在第12章详细介绍这些问题，构建这个简单的操作系统以作为计算机系统的最后一个模块。

9.5 项目 Project

对象 本项目意在让读者了解Jack语言，基于两个目的：即将在项目10和项目11中编写Jack编译器，在项目12中编写Jack操作系统。

约定 改编或者开发一个应用程序，比如简单的计算机游戏或某种交互式程序。设计构建该应用程序。

资源 在本项目中需要三个工具：Jack 编译器，用来将你的程序翻译成一组 .vm 文件；VM 仿真器，用来运行和测试翻译后的文件；Jack 操作系统。

Jack OS Jack 操作系统以一系列 .vm 文件的形式来表现并发挥作用。这些文件构成了 Jack 编程语言标准库的实现。为了使任何 Jack 程序能够被正确执行，编译后的 .vm 文件必须与 Jack OS 的所有 .vm 文件保存在相同路径下。当 Jack OS 检测到 OS 相关的错误时，会显示数字错误代码（而不是显示文本，因为文本会比数字占用更多内存空间）。在文件 projects/09/OSerrors.txt 中列出了所有 Jack OS 支持的错误代码及其描述信息。

编译并运行 Jack 程序

0. 程序必须被保存在独立的路径下，比如 Xxx 。首先要创建该路径，然后将 tools/OS 中的所有文件拷贝到该路径下。

1. 编写你的 Jack 程序，该程序可以由一个或多个 Jack 类组成，每个类都保存在独立的 .jack 文本文件中（文件名必须是类的名称）。把这些 .jack 文件全都置于 Xxx 路径下。

2. 利用提供的 Jack 编译器来编译程序。最好将编译器应用到程序路径的名字（Xxx）中。这会使得编译器将该路径下所有的 .jack 类翻译成相应的 .vm 文件。如果编译中报告了编译错误，请调试程序，然后重新编译 Xxx 直到没有任何错误信息为止。

3. 程序路径目前应该包含三组文件：（1） .jack 源文件；（2）编译后的 .vm 文件，与每个 .jack 类文件一一对应；（3）其他 .vm 文件，包含 Jack OS 。为了测试编译后的程序，调用 VM 仿真器，加载整个 Xxx 程序路径。然后运行程序。如果遇到运行期错误或者异常的程序行为，请修改程序然后返回到第 2 步。

Jack 程序范例 与本书配套的软件包提供了完整的 Jack 应用范例，保存在 projects/09/Square 路径下。该路径包含三个类的 Jack 源代码，组成一个简单的交互式小游戏。

第 10 章　编译器 I：语法分析

Compiler I: Syntax Analysis

Neither can embellishments of language be found without arrangement and expression of thoughts, nor can thoughts be made to shine without the light of language.
既未组织思想又未表达思想，语言便黯淡无光；没有语言之光芒照耀，思想便无法闪光。
——Cicero（公元前 106～公元前 43），罗马作家、政治家、演说家

前一章介绍了简单的基于对象的 Jack 编程语言，语法与 Java 和 C#的语法类似。本章开始为 Jack 语言构建**编译器**。编译器是一种程序，能将高级语言程序从源语言翻译成目标语言。这个翻译过程（即**编译**，*compilation*）从概念上讲由两个不同的任务组成。首先，我们必须理解源程序的**语法**（*syntax*），以此来揭示程序的**语义**（*semantics*）。比如，对代码进行语法分析可得知程序想要声明数组或操纵对象。该信息让我们能够使用目标语言的语法来重构程序的逻辑。第一个任务通常称为**语法分析**（*syntax analysis*），会在本章进行阐述；第二个任务，即**代码生成**（*code generation*），会在第 11 章中介绍。

怎么能断定编译器是否能够“理解”语言的语法呢？其实，只要编译器生成的代码能执行期望的任务，就该乐观地认为该编译器执行了正确操作。然而本章仅构建编译器的**语法分析器**（*syntax analyzer*）模块，它并不具备代码生成功能。如果希望单独测试该语法分析器，就必须设计某种可行的方法来证明它“理解了”源程序。我们的方法是让语法分析器输出 XML 文件，其格式反映了输入程序的语法结构。通过观察生成的 XML 输出，应该能够确定分析器是否正确地解析了输入程序。

本章的**背景知识**部分介绍了构建语法分析器所必需的基本概念：词法分析（lexical analysis）、上下文无关语法、语法分析树（parse trees），以及用于分析它们的递归下降（recursive-descent）算法。这些为**规范详述**部分提供了基础知识，该部分介绍 Jack 语言的

正式语法和 Jack 分析器的输出格式。**实现**部分介绍了用于构建 Jack 分析器的软件架构以及 API。跟前面章节一样，**项目**部分给出了逐步操作指示和测试程序，用于实际构建并测试语法分析器。在下一章，该分析器将被扩展成完整的编译器。

从零开始编写编译器是一项复杂的任务，涉及计算机科学里面很多基本主题。它需要对语言翻译技术和语法分析技术的理解，需要使用树和哈希表等一些经典数据结构，需要应用一些高效的递归编译算法。因此，编写编译器也是富有挑战性的任务。通过将编译器的构建分为两个独立的项目（或者说是四个——包括两个 VM 项目在内），就能够进行模块化的开发并且对各模块进行独立的单元测试（unit-testing），从而使编译器的构建变得相对独立且易于管理。

为什么要不嫌麻烦地构建编译器呢？首先，对编译过程内部原理的掌握将会使你成为更好的高级程序员。其次，用于描述编程语言的规则和语法，同样也能用于描述不同应用（从计算机制图到数据库管理，再到通信协议，以至生物信息学）中的数据集的语法。因此，虽然大多数程序员在编程生涯中不必开发编译器，但很有可能需要分析和处理某种复杂语法的文件。这些任务与分析编程语言一样，采用相同的概念和技术，本章正是要阐述这些内容（见图 10.1）。

10.1 背景知识 Background

典型的编译器由两个主要模块组成：**语法分析**（*syntax analysis*）模块和**代码生成**（*code generation*）模块。语法分析任务通常可以进一步分成两个模块：**字元化**（*tokenizing*）模块将输入的字符分组成语言原子元素（language atoms）；然后由**语法分析**（*parsing*）模块将所得到的语言原子元素集合同语法规则相匹配。注意，这些行为相对于目标语言是完全独立的。因为在本章并不涉及代码生成，所以我们让语法分析器输出 XML 文件，表示程序经编译后的语法分析结构。这么做有两点好处。首先可以使用任何 Web 浏览器来读取这个 XML 文件，以此来证明语法分析器正确地解析了源程序。其次，为了输出文件，就

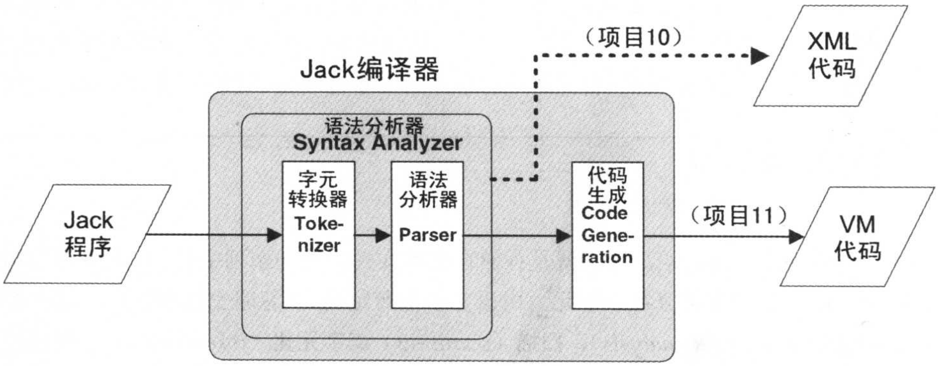


图 10.1 Jack 编译器。项目 10 是中间步骤，用来进行编译器的阶段化开发以及语法分析器模块的单元测试

必须依据特定的软件架构来编写语法分析器，而该架构在后续内容中会演化成完整的编译器架构。下一章用生成可执行 VM 代码的子程序来替换生成 XML 代码的程序即可，而不用去修改编译器架构的其他部分。

本章集中讨论编译器的**语法分析**（*syntax analyzer*）模块，其任务是“理解高级源程序的结构。”在这里需要对这句话做一些解释。人们在阅读计算机程序时，立即就能弄清程序的结构。人们能辨认类（class）和方法（method）在哪里开始，到哪里结束，哪些是声明，哪些是语句，哪些是表达式，它们是怎么组织的，等等。这种理解不无价值，因为这需要具有对嵌套模式的辨认能力和归类能力：在一般程序中，类包含方法，方法包含语句，语句包含其他语句，其他语句中又包含表达式，等等，都是这样的嵌套模式。为了正确地弄清语言结构，人们必须递归地将其映射到语言语法所允许的文本范围之内。

如果要理解英语这样的一门自然语言，那么关于语法规则如何在人类大脑中表示，以及这些规则是先天具备的还是后天形成的这类问题，就会引起激烈争辩。然而，如果只考虑**形式语言**（因其简单性而几乎不配被称为“语言”），就能准确地知道如何将其语法结构

形式化。我们通常使用一组称为**上下文无关语法**（*context-free grammar*）的规则来描述编程语言。要理解（即解析）给定的程序，就意味着要决定程序文本和语法规则之间的准确对应。为了做到这一点，首先必须将程序的文本转换成一系列**字元**（*token*），如下所述。

10.1.1 词法分析 Lexical Analysis

程序最简单的语法形式就是存储在文本文件中的一系列字符。对程序进行语法分析的第一步，是将字符分组成**字元**（由语言语法所定义），忽略空格和注释。这一步通常称为**词法分析**（*lexical analysis*）、**扫描**（*scanning*）或**字元化**（*tokenizing*）。一旦程序被分组成字元，字元（而不是字符）就被看作程序的基本原子，字元集合就成为编译器的输入。图 10.2 举例说明了 C 语言或者 Java 语言的典型代码段的字元转换结果。

如图 10.2 所示，字元包括不同的种类（或称类型），比如 `while` 是**关键字**（*keyword*），`count` 是**标识符**（*identifier*），`<=` 是**操作符**（*operator*），等等。编程语言通常都会指定其所允许的字元类型，以及用于将字元组合成正确程序结构的准确语法规则。比如，一些语言规定“++”为合法的操作符字元，而其他语言可能不这样规定，有的编译器会认为两个连续“+”字符组成的标记是不合法的。

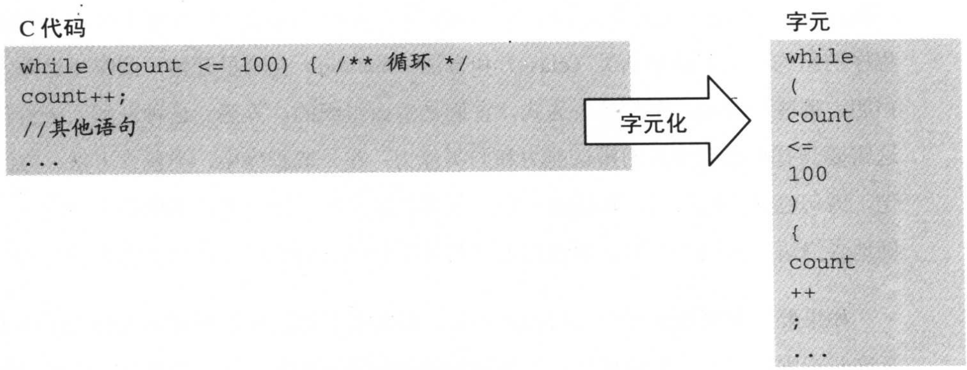


图 10.2 词法分析（lexical analyzer）

10.1.2 语法 Grammars

一旦对程序进行语法分析，形成一系列字元（即字元流，token stream），就面临更具挑战性的任务，即将字元进行分析，理解成对应的规范结构。换句话说，必须想办法考虑如何将这些字元分组成变量声明、语句、表达式等等的语言结构。通过将字元集合按照预定义的规则集（set of rules），即**语法**（*grammar*）进行组合匹配，就可以实现分组和分类任务。

几乎所有的编程语言，以及大多数用于描述复杂文件类型语法的其他的形式语言，都可以使用**上下文无关语法**（*context-free grammars*）来描述。上下文无关语法是一组规则，用来指定语言中的语法元素如何由更简单的元素组成。比如，Java 语法允许将基本元素 `100`、`count` 和`<=`联合成表达式 `count<=100` 。同样的，Java 语法允许确定文本 `count<=100` 是正确的 Java 表达式。事实上，每种语法都可以从两个方面来考察。从声明的观点来看，语法规定了可行的方案，来将字元（token，也称为**终结符**，*terminals*）组合成更高级的语法元素（也称为**非终结符**，*non-terminals*）。从分析的观点来看，语法是用于执行反向任务的规则：将给定的输入，即通过字元化阶段（tokenizing phase）得到的字元集合，解析成非终结符、较低级的非终结符，以及最后不能继续分解的终结符。图 10.3 给出了典型语法的例子。

本章使用这样的表示法来描述语法：终结符用单引号括起来的粗体字来表示，非终结符用一般字体表示。当有不止一种方法来分析非终结符时，就使用“|”符号列出可能的结果。因此，图 10.3 指定 *statement*（语句）可以是 `while` 语句，或者 `if` 语句，等等。语法规则一般是高度递归的，图 10.3 所示的也不例外。比如，*statementSequece*（语句序列）可以是空的，或者是单一的 *statement* 隔一个分号之后又接着一个 *statementSequece*。这个递归的定义可以容纳一系列数量为 0、1、2 或任意正数个由分号分隔的语句。作为练习，读者可用图 10.3 来确认图中右边的文本是否组成了正确的 C 语言代码。你可以试着用 *statement* 来匹配整个文本。

10.1.3 语法分析 Parsing

检查语法是否将所输入的文本看作合法输入，这个过程称为**语法分析**（*parsing*）。正如前面提到的，“分析给定的文本”意味着，决定文本和给定语法规则之间的一致性。因

```
statement: whileStatement
           | ifStatement
           | ... // 其他可能的语句
           | '{' statementSequence '}'

whileStatement: 'while' '(' expression ')'
                statement

ifStatement: ... // "if"的定义

statementSequence: '' // 空序列(null)
                  | statement ';'
                  statementSequence

expression: ... // "expression"表达式定义
...            // 更多定义
```

```
while (expression) {
   statement;
   statement;
   while (expression) {
     while(expression)
        statement;
     statement;
   }
}
```

图 10.3 C 语言语法的子集（左边部分）与符合该语法的一段范例代码（右边部分）

为语法规则是分层的，所以语法分析器（parser）生成的输出可以用称为**语法分析树**（*parse tree*）或**导出树**（*derivation tree*）的树状数据结构来描述。图 10.4 中给出了典型的例子。

需要注意的是，作为语法分析过程的副作用，输入文本的整个语法结构被揭示出来。某些编译器用显式的数据结构（可以更进一步用于代码生成和错误报告）来表示这棵树。而其他一些编译器（包括将要构建的 Jack 编译器）则隐式地表示程序结构，即时地进行代码生成和错误报告——这样的编译器不需要在内存中保存整个程序结构，仅仅保存与当前被分析的元素相关的子树。更多相关内容会在后面介绍。

递归下降分析（Recursive Descent Parsing） 有很多算法可用来构建语法分析树。自顶向下的方法，也称为**递归下降分析**，是应用由语法规则描述的嵌套结构来尝试递归地分析字元集合形成的输入流（token stream）。现在来考虑如何编写实现该功能的**语法分析器程序**（*parser program*）。对于语法中每个描述非终结符的规则，可以为语法分析器程序配备递归子程序，该子程序用来递归地分析非终结符。如果非终结符仅由终结符原子构成，该程序就很容易处理它们。否则，对于规则右边的每个非终结符构建块（building block），程序可以递归地调用“分析该非终结符”的程序。整个处理过程递归进行，直到所有终结符原子元素全部被处理完毕。

C代码

```
while (count<=100) {
  count++;
// ...
```

部分 C 语言语法

```
statement: whileStatement | ifStatement
         | ... | '{' statementSequence '}'
whileStatement: 'while' '(' expression ')'
                     statement
ifStatement: ... // "if"的定义
statementSequence: '' // Null
         | statement ';' statementSequence
expression: ... // 表达式的定义
```

字元转换
（语法分析器的输入）：

```
while
(
count
<=
100
)
{
count
++
;
...
```

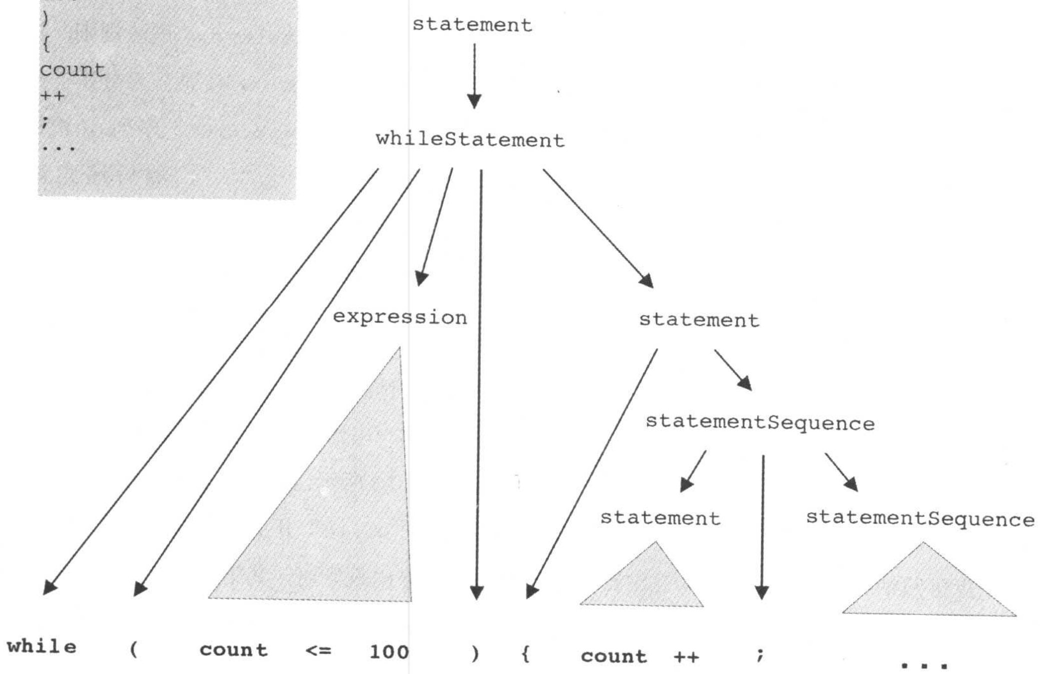


图 10.4 某程序代码段依照语法生成的语法分析树。灰色角形表示更底层的语法分析树

举例来说，假设要编写遵循图 10.3 中语法的递归下降分析器。由于该语法有 5 个导出规则，因此该分析实现也可以由 5 个程序组成：`parseStatement()`、`parseWhileStatement()`、`parseIfStatement()`、`parseStatementSequence()` 和 `parseExpression()`。这些程序的语法分析逻辑应该遵循那些“出现在相应语法规则右边”的语法模式。因此 `parseStatement()`应该从确定输入中的第一个字元来开始它的处理过程。在确定了字元的类型后，子程序就能确定在处理哪一种类型的表达式，然后调用与该表达式型相匹配的分析程序。

例如，如果输入流如图 10.4 中所描述的，程序会确定第一个字元是 `while`，然后调用相应的 `parseWhileStament()`子程序。根据相应的 `whileStatement` 语法规则，该程序接下来应该读取终结符“`while`”和“`(`”，然后调用 `parseExpression()`来分析非终结符 *expression*（表达式），在该函数返回后（本例中分析了“`count<=100`”序列），根据语法规定，这时 `parseWhileStament()`读取的终结符应该是“`)`”，然后递归调用 `parseStatement()`。该调用将会递归进行下去，直到读取的字元都是终结符为止。显然，同样的逻辑也可以用来检查源程序中的语法错误。编译器越好，错误诊断的能力就越强。

LL(0)语法 递归分析算法是简单高效的算法。唯一可能的复杂性在于分析的非终结符有多种可选的解释。例如，当 `parseStatement()`试图分析一条语句时，它预先不知道该语句是 while-语句、if-语句还是用花括号括起来的一串语句。可能性的范围由语法来决定，在某些情况下很容易辨别是哪一种选择。比如图 10.3 所示，如果第一个字元是“`while`”，显然碰到的是 while-语句，因为它是语法规则中以“`while`”开头的唯一语句。我们的观点可归纳如下：每当一个非终结符有多种可选择的导出规则时，其第一个字元足以确定该非终结符所属的表达式类型，而不会出现不确定的情况。具有这种属性的语法称为 *LL(0)*。递归下降算法可以简洁明了地处理这类语法。

当第一个字元不足以确定元素的类型时，对下一个字元的“提前查看”有可能可以解决这种不确定性。这种分析显然是可以做到的，但当我们需要顺着字元流去读取越来越多的字元时，分析就变得复杂起来。这里要介绍的 Jack 语言语法，就几乎是 *LL(0)*，因此它可以通过递归下降分析器来（recursive descent parser）处理。唯一的例外就是表达式的分析，在必要时需要“提前查看”后续字元。

10.2 规范详述 Specification

这一节分为两个独立的部分。首先介绍 Jack 语言的**语法**。然后介绍**语法分析器**（*syntax analyzer*），根据语法来对程序进行语法分析（parse）。

10.2.1 Jack 语言语法 The Jack Language Grammar

第 9 章中给出的 Jack 语言的功能规范主要面向 Jack 程序员。现在转到给出该语言的形式规范（formal specification），该规范主要面向 Jack 编译器开发人员。语法规范有如下约定：

'**xxx**'：单引号括起来的黑体，用于逐字出现的字元（“终结符”）；
xxx：常规字体，用于语言构造的名字（“非终结符”）；；
( )：圆括号，用于语言构词的分组；
x|y：指示 x 或者 y 将出现；
x?：指示 x 出现 0 或者 1 次；
x*：指示 x 出现 0 或者多次。

图 10.5 给出了遵循这些约定的 Jack 语言语法。

10.2.2 Jack 语言的语法分析器 A Syntax Analyzer for the Jack Language

语法分析器（syntax analyzer）的主要目的是读取 Jack 程序，根据 Jack 语法“理解”其语法结构。“理解”的意思是语法分析器在处理过程中的每一步都必须确定当前读取的程序要素是 *expression* 或 *statement* 或 *variable name*，等等。语法分析器必须以完全递归的

语义要素:	Jack 语言包括 5 种终结元素（即字元）:
关键字:	'class' | 'constructor' | 'function' | 'method' | 'field' | 'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' | 'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' | 'while' | 'return'
符号:	'{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'
integerConstant（整数常量）:	0 至 32767 的十进制数。
StringConstant（字符串常量）	'"' 一系列 Unicode 字符（不包括双引号或新行符） '"'
identifier（标识符）:	由一系列字母、数字、下划线（'_'）组成，不能以数字开头。
程序结构:	Jack 程序是类（classes）的集合，每个类单独存放在一个文件当中。类（class）是 Jack 程序的编译单元（compilation unit）。类是根据下列的上下文无关语法字元构成的序列:
类:	'class' className '{' classVarDec* subroutineDec* '}'
classVarDec:	('static' | 'field') type varName (',' varName)* ';'
类型:	'int' | 'char' | 'boolean' | className
subroutineDec:	('constructor' | 'function' | 'method') ('void' | type) subroutineName '(' parameterList ')' subroutineBody
parameterList:	((type varName) (',' type varName)*)?
subroutineBody:	'{' varDec* statements '}'
varDec:	'var' type varName (',' varName)* ';'
className:	标识符
subroutineName:	标识符
varName:	标识符

图 10.5 Jack 语言的完整语法

<table>
<tr><td colspan="2">语句:</td></tr>
<tr><td>statements:</td><td>statement*</td></tr>
<tr><td>statement:</td><td>letStatement | ifStatement | whileStatement |
doStatement | returnStatement</td></tr>
<tr><td>letStatement:</td><td>'let' varName ('[' expression ']')? '=' expression ';'</td></tr>
<tr><td>ifStatement::</td><td>'if' '(' expression ')' '{' statements '}'
('else' '{' statements '}')?</td></tr>
<tr><td>whileStatement:</td><td>'while' '(' expression ')' '{' statements '}'</td></tr>
<tr><td>doStatement:</td><td>'do' subroutineCall ';'</td></tr>
<tr><td>ReturnStatement</td><td>'return' expression? ';'</td></tr>
<tr><td colspan="2">表达式:</td></tr>
<tr><td>expression:</td><td>term (op term)*</td></tr>
<tr><td>term:</td><td>integerConstant | stringConstant | keywordConstant |
varName | varName '[' expression ']' | subroutineCall |
'(' expression ')' | unaryOp term</td></tr>
<tr><td>subroutineCall:</td><td>subroutineName '(' expressionList ')' | (className |
varName) '.' subroutineName '(' expressionList ')'</td></tr>
<tr><td>expressionList:</td><td>(expression (',' expression)*)?</td></tr>
<tr><td>op:</td><td>'+'|'-'|'*'|'/'|'&'|'|'|'<'|'>'|'='</td></tr>
<tr><td>unaryOp:</td><td>'-'|'~'</td></tr>
<tr><td>KeywordConstant:</td><td>'true'|'false'|'null'|'this'</td></tr>
</table>

续图 10.5 Jack 语言的完整语法

形式来保存语法结构的信息。没有此信息，就不可能进入下一步的代码生成阶段（代码生成是整个编译器项目的最终目标）。

为了证明语法分析器能够“理解”输入的程序结构，可以让其输出处理后的源程序文本，该文本具有良好的结构，并且容易阅读。当然读者可以考虑其他的方法来证明。本书决定让语法分析器输出 XML 文件，该文件的 XML 标记语言格式反映了程序的语法结构。通过观察该 XML 输出文件（可以利用 Web 浏览器来进行查看），我们应该能够立即确认该语法分析器是否正确执行了任务。

10.2.3 语法分析器的输入 The Syntax Analyzer's Input

Jack 语法分析器接受单一的命令行参数，如下所示：

```
prompt> JackAnalyzer source
```

source 是形如 `Xxx.jack`（扩展名是必要的）的文件名称或是包含若干个 `.jack` 文件的路径名（在这种情况下，没有扩展名）。语法分析器将 `Xxx.jack` 文件编译成名为 `Xxx.xml` 的文件，保存在与源文件相同的路径下。如果 *source* 是路径名，那么该路径下的每个 `.jack` 文件都将被编译，分别生成相应的 `.xml` 文件，保存在相同路径下。

每个 `Xxx.jack` 文件都是字符流（即一系列字符）。根据 Jack 语言的**语义元素**（*lexical elements*）所指定的规则（如图 10.5 的上部），该字符流应该被字元化，形成字元流（即一系列字元）。这些字元可以用任意数量的空格符、换行符和注释（这些符号都被忽略）分隔开来。注释采用下面的几种标准形式：`/*注释*/`，`/**API 注释*/`，以及 `//到行末的注释`。

10.2.4 语法分析器的输出 The Syntax Analyzer's Output

前面介绍过，Jack 编译器的开发分为两个阶段（见图 10.1），首先是语法分析器（syntax

analyzer)。本章让语法分析器输出关于输入程序的 XML 文件，图 10.6 给出了例子。为了做到这一点，语法分析器必须辨识语言构造中的两个主要类型：终结符和非终结符。这些构词按如下方式来处理。

非终结符 任何时候遇到形如 *xxx* 的非终结语言元素，语法分析器应该生成 XML 输出：

```
<xxx>
    xxx 元素部分的递归代码。
</xxx>
```

这里 *xxx* 是（只能是）下列 Jack 语法的非终结符之一：

- class, classVarDec, subroutineDec, parameterList, subroutineBody, varDec;
- statements, whileStatement, ifStatement, returnStatement, letStatement, doStatement;
- expression, term, expressionList。

终结符 任何时候遇到形如 *xxx* 的终结语言元素，语法分析器应该生成 XML 输出：

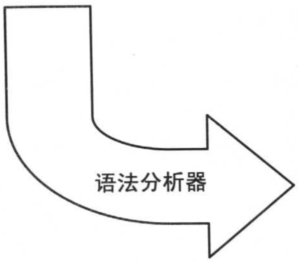
```
<xxx>终结符</xxx>
```

这里 *xxx* 是 Jack 语言所认可的 5 种字元类型之一（见 Jack 语法的"语义元素（lexical elements）"部分的描述），也就是：keyword, symbol, integerConstant, stringConstant, 或者 identifier。

图 10.6 中显示了分析器的输出，观察该输出文本，可能会有似曾相识的感觉。在本章前面部分提到过，程序的结构可以被分析成**语法分析树**（*parse tree*）。事实上，XML 输出就是关于这种分析树的文本描述。在语法分析树中，非终结节点（non-terminal nodes）形成了一种"超结构（super structure）"，该结构描述了树的终结节点（terminal nodes，即字元）是如何被组织成语言构词的。这种模式被反映到 XML 的输出里，在输出中，非终结 XML 元素描述了终结 XML 项是如何组织的。按照同样的方式，字元生成机制生成的终结符（字元）形成了 XML 输出的最底层，正如它们形成程序语法分析树的终端叶子节点一样。

语法分析器的输入（Jack 代码）

```
Class Bar {
  method Fraction foo(int y) {
    var int temp; // 一个变量
    let temp = (xxx+12)*-63;
    ...
```



语法分析器的输出（XML 代码）

```
<class>
  <keyword> class </keyword>
  <identifier> Bar </identifier>
  <symbol> { </symbol>
  <subroutineDec>
    <keyword> method </keyword>
    <identifier> Fraction </identifier>
    <identifier> foo </identifier>
    <symbol> ( </symbol>
    <parameterList>
      <keyword> int </keyword>
      <identifier> y </identifier>
    </parameterList>
    <symbol> ) </symbol>
    <subroutineBody>
      <symbol> { </symbol>
      <varDec>
        <keyword> var </keyword>
        <keyword> int </keyword>
        <identifier> temp </identifier>
        <symbol> ; </symbol>
      </varDec>
      <statements>
        <letStatement>
          <keyword> let </keyword>
          <identifier> temp </identifier>
          <symbol> = </symbol>
          <expression>
          ...
          </expression>
          <symbol> ; </symbol>
          ...
```

图 10.6　Jack 分析器工作实例

代码生成 刚刚描述了分析器的 XML 输出。在下一章中，将用能够生成可执行 VM 代码的子程序来替换这个输出 XML 的子程序，以此来得到完整的 Jack 编译器。

10.3 实现 Implementation

第 10.2 节给出了构建 Jack 语言的**语法分析器**（*syntax analyzer*）所需的全部信息，但是没有给出任何实现细节。本节将给出语法分析器的软件架构。建议将实现分成三个模块：

- `JackAnalyzer`： 建立和调用其他模块的顶层驱动模块；
- `JackTokenizer`： 字元转换器（字元化）模块；
- `CompilationEngine`： 自顶向下的递归语法分析器。

这些模块用来处理语言的语法。在下一章里，将用**符号表**（*symbol table*）和 **VM 代码生成器**（*VM-code writer*）这两个附加模块来扩展整个体系，以此来完成 Jack 语言编译器的完整构建工作。因为本项目中驱动分析过程的模块也驱动下一个项目中的整个编译，所以该模块也称为 `CompilationEngine`（编译引擎）。

10.3.1 *JackAnalyzer* 模块 The *Jack Analyzer* Module

分析器程序对给定的 *source* 进行操作，*source* 是形如 `Xxx.jack` 的文件名称或者包含若干个此类文件的路径名。对于每个源 `Xxx.jack` 文件，分析器按如下逻辑进行处理：

1. 从 `Xxx.jack` 输入文件创建 *JackTokenizer*。
2. 创建名为 `Xxx.xml` 的**输出文件**，准备写文件。
3. 使用 *CompilationEngine* 来将输入的 *JackTokenizer* 编译成**输出文件**。

10.3.2 *JackTokenizer* 模块
The *Jack Tokenizer* Module

JackTokenizer：从输入流中删除所有的注释和空格，并根据 Jack 语法的规则将输入流分解成 Jack 语言的字元（终结符）。

程 序	参 数	返回值	功 能
Constructor	输入文件/输入流	—	打开输入文件/输入流，准备进行字元转换操作
hasMoreTokens	—	布尔值	输入中是否还有字元？
advance	—	—	从输入中读取下一个字元，使其成为当前字元。该函数仅当 hasMoreTokens()返回为真时才能调用。最初始状态是没有当前字元
tokenType	—	KEYWORD, SYMBOL, IDENTIFIER, INT-CONST, STRING_CONST	返回当前字元的类型
keyword	—	CLASS, METHOD, INT, FUNCTION, BOOLEAN, CONSTRUCTOR, CHAR, VOID, VAR, STATIC, FIELD, LET, DO, IF, ELSE, WHILE, RETURN, TRUE, FALSE, NULL, THIS	返回当前字元的关键字。仅当 tokenType() 返回值为 KEYWORD 时才被调用
symbol	—	字符	返回当前字元的字符。仅当 tokenType() 的返回值为 SYMBOL 时才被调用
identifier	—	字符串	返回当前字元的标识符。仅当 tokenType() 的返回值为 IDENTIFIER 时才被调用
intVal	—	整数值	返回当前字元的整数值。仅当 tokenType() 的返回值为 INT_CONST 时才被调用
stringVal	—	字符串	返回当前字元的字符串值，没有双引号。仅当 tokenType()的返回值为 STRING_CONST 时才被调用

10.3.3 *CompilationEngine* 模块
The *CompilationEngine* Module

CompilationEngine：执行编译输出。从 `JackTokenizer` 中得到输入，然后将分析后的结果放入输出文件或输出流。输出是通过一系列 `compilexxx()` 子程序生成的，每个子程序对应 Jack 语法中的一个语法要素 xxx（可以是字元或构成语句的一般符号）。这些程序之间约定如下：每个 `compilexxx()` 程序应该从 `CompilationEginge` 的输入中读取语法要素 xxx，利用 `advance()` 函数取出当前要素 xxx 的下一个要素，并输出当前要素 xxx 的分析结果。因此，仅当下一个要素是 xxx 时，才会调用该要素对应的分析函数 `compilexxx()`。

在本章构建的编译器中，该模块生成结构化的代码，再用 XML 标记（tags）对代码进行包装。在下一章构建的完整编译器中，该模块生成可执行的 VM 代码。两者的分析逻辑和 API 模块是完全一样的。

程　序	参　数	返回值	功　能
构造函数	输入流/ 输入文件 输出流/ 输出文件	—	利用给定的输入和输出创建新的编译引擎。接下来必须调用 `compileClass()`
`CompileClass`	—	—	编译整个类
`CompileClassVarDec`	—	—	编译静态声明或字段声明
`CompileSubroutine`	—	—	编译整个方法、函数或构造函数
`compileParameterList`	—	—	编译参数列表（可能为空），不包含括号“( )”

续表

程 序	参 数	返回值	功 能
compileVarDec	—	—	编译 Var 声明
compileStatements	—	—	编译一系列语句，不包含大括号“{ }”
compileDo	—	—	编译 do 语句
compileLet	—	—	编译 let 语句
compileWhile	—	—	编译 while 语句
compileReturn	—	—	编译 return 语句
compileIf	—	—	编译 if 语句，包含可选的 else 从句
CompileExpression	—	—	编译一个表达式
CompileTerm	—	—	编译一个 “*term*”。本程序在 “从多种可能的分析规则中作出决策” 的时候会遇到一点难度。特别是，如果当前字元为标识符，那么本程序就须要区分变量、数组、子程序调用这三种情况。通过提前查看下一个字元（可以为 “[”、“(” 或 “.”），就可以区分这三种可能性了。后续任何其他字元都不属于这个 term，故不须要取用
CompileExpressionList	—	—	编译由逗号分隔的表达式列表（可能为空）

10.4 观点
Perspective

虽然使用语法分析树和 XML 文件能很方便地描述计算机程序的结构，但有必要知道：编译器并不需要显式地维护该数据结构。比如，本章描述的语法分析算法是“实时”的，这意味着在读取输入时就立即进行分析，在内存中并不存储编译器输入的整个源程序。有两种方法可以实现这样的分析。较简单的方法是本章中介绍的自顶向下的方法。较高级的方法是自底向上的方法，因其涉及到大量额外的理论知识，故在此不作介绍。

事实上，这一章几乎回避了所有在典型编译课程中研究的形式语言理论，这是因为我们为 Jack 语言规定了非常简单的语法，回避了这些问题。利用递归下降技术可以很容易地编译 Jack 语法。比如，Jack 语法并没有规定在表达式计算中的常规运算符的优先级（乘法优于加法，等等）。这使得我们回避了比本章介绍的自顶向下分析算法功能更强大，然而技术更复杂的分析算法。

本章中另一个几乎没有提到的主题是如何描述语言的语法。有个称为**形式语言**（*formal languages*）的丰富理论，它讨论了各类语言的属性，包括元语言（metalanguages）及其描述形式。这也是计算机科学与人类语言的交叉点，形成了交叉研究学科——**计算语言学**（*computational linguistics*）。

最后有必要提到的是，语法分析器并不是独立的程序，人们很少从头凭空开始编写语法分析器。换言之，程序员们通常使用各种“生成编译器”的工具来构建字元转换器（tokenizers）和分析器（parsers），这类工具如 *LEX*（即 *LEXical analysis*）和 *YACC*（即 *Yet Another Compiler Compiler*）。这些工具接收上下文无关语法作为输入，生成法分析代码（能够对按此语法编写的程序进行字元转换和语法分析）作为输出。然后，就可以对生成的代码进行定制，以满足不同应用的特定编译需求。遵循本书中一贯坚持的“show me（让我看看）”之精神，我们在编译器的实现中不使用这样的黑盒子，而是从头开始去构建所有部分。

10.5 项目 Project

编译器的构建涵盖两个项目：项目 10 和项目 11。这一节介绍如何构建本章所描述的语法分析器。在下一章中，将这个分析器扩展成完整的 Jack 编译器。

目标 根据 Jack 语法构建用于分析 Jack 程序的语法分析器。分析器的输出应该是 10.2 节中定义的 XML 文件。

资源 本项目中的主要工具是用于实现语法分析器的编程语言。还将用到与本书配套的 `TextComparer` 工具，它能够将语法分析器生成的输出文件和我们提供的比较文件进行比较。你可能还会使用到 XML 阅读器（可以是任何标准的 Web 浏览器）来读取生成的输出文件和本书提供的文件。

约定 分两个阶段来编写语法分析器程序：字元化（tokenizing，或称字元转换）阶段和语法分析（parsing）阶段。使用该程序来分析这里提到的所有 `.jack` 文件。对每个源 `.jack` 文件，分析器应该生成 `.xml` 输出文件。生成的文件应该和我们提供的 `.xml` 文件内容相同。

测试程序

语法分析器的工作就是分析用 Jack 语言编写的程序。因此，测试分析器是否正确的方法是，让分析器分析几个典型的 Jack 程序。我们提供了两个这样的测试程序，*Square Dance* 和 *Array Test*。前者包含了 Jack 语言除数组处理之外的所有特性，而后者包含数组处理，并提供了 *Square Dance* 程序的**简化版本**，在下面会介绍。

本书提供了这三个程序所有的 Jack 源文件。对于每个 `Xxx.jack` 文件，提供了两个名为 `XxxT.xml` 和 `Xxx.xml` 的比较文件。这些文件包含了**字元转换器**（*tokenizer*）和**语法分析器**（*parser*）处理 `Xxx.jack` 文件之后产生的输出。

- *Square Dance*（`projects/10/Square`）：小型交互式游戏，允许用户使用键盘上的四个方向键在屏幕上移动一个黑色方块。

- 无表达式的 *Square Dance*（projects/10/ExpressionlessSquare）：与 *Square Dance* 几乎相同的版本，唯一的区别是原始程序的每个表达式在这里被简单的标识符（某个变量名）所替代。例如，Square 类中有个方法能把图形化方块对象的尺寸增加 2 个像素，只要增加后的尺寸不使得方块图形溢出屏幕的边界就行了。这个方法的代码如下所示：

Square 类代码

```
method void incSize() {
  if (((y + size < 254) &
     ((x + size <510) {
    do erase();
    let size = size + 2;
    do draw();
  }
  return;
}
```

ExpressionSquare 类代码

```
method void incSize() {
  if (x) {
    do erase();
    let size = size;'
    do draw();
  }
  return;
}
```

注意，用变量替换表达式得到的程序无法被与本书配套的 Jack 编译器编译。然而，它符合所有的 Jack 语法规则。无表达式类文件具有和原始版本的文件相同的名字，但是它们存放在各自独立的路径下。

- *Array Test*（Projects/10/ArrayTest）：只包含一个类（class）的 Jack 程序，计算用户提供的一串整数的平均值，这些整数使用了数组的概念和数组的处理方式。

利用测试程序进行实验 如果有兴趣，你可以使用本书提供的 Jack 编译器来编译 *Square Dance* 和 *Array Test* 程序，然后使用提供的 VM 仿真器来运行编译后的代码。这些操作与本项目完全无关，其主要目的是为了强调这些测试程序不仅仅是纯文本。

第 1 阶段：字元转换器

首先，实现 10.3 节中描述的 JackTokenizer 模块。用其处理包含 Jack 代码的文本文件时，字元转换器应该产生一串字元，每个字元都应该跟它的分类名称一起被列印在单独

的行上：*symbol*（符号），*keyword*（关键字），*identifier*（标识符），*integer constant*（整数常量），或者 *string constant*（字符串常量）。分类名应该使用 XML 标记来记录。这里给出了一个例子：

源代码

```
if (x < 153)
   {let city="Paris";}
```

字元转换器的输出

```
<tokens>
   <keyword> if </keyword>
   <symbol> ( </symbol>
   <identifier> x </identifier>
   <symbol> &lt; </symbol>
   <integerConstant> 153 </integerConstant>
   <symbol> ) </symbol>
   <symbol> { </symbol>
   <keyword> let </keyword>
   <identifier> city </identifier>
   <symbol> = </symbol>
   <stringConstant> Paris
   </stringConstant>
   <symbol> ; </symbol>
   <symbol> } </symbol>
</tokens>
```

注意到在 *string Constant* 情况下，字元转换器有意地去掉了双引号字符。

字元转换器的输出具有两个由 XML 规范导致的“特点”。首先，XML 文件必须用开始/结束标记来包围整个文本内容，因此`<tokens>`和`</tokens>`被追加到输出中。其次，Jack 语言中用到的 3 种符号（`<`，`>`，`&`）也是 XML 标记所采用的字符，因此它们不能以数据形式出现在 XML 文件中。为了解决这个问题，需要字元转换器分别以`<`、`>`和`&amp`来输出这 3 个字元。比如，为了使得文本“`<symbol> < </symbol>`”能够在 Web 浏览器中正确地显示，源 XML 应该被写成“`<symbol> < </symbol>`”。

测试你的字元转换器

- 用 *Square Dance* 和 *Test Array* 程序来测试你的字元转换器。不必使用 *Square Dance* 的无表达式版本来进行测试。

- 对于每个源文件 `Xxx.jack`，让你的字元转换器将输出文件命名为 `XxxT.xml`。将你的字元转换器应用到测试程序中的每个类文件，然后使用提供的 `TextComparer` 工具来比较生成的输出文件和提供的 `.xml` 比较文件。
- 因为你的字元转换器生成的输出文件将会有与提供的比较文件相同的名称和扩展名，所以我们建议将它们放在另外单独的路径下。

第 2 阶段：语法分析器

接下来，实现 10.3 节中描述的 `CompilationEngine` 模块。按照 API 的规范编写其中的每个方法，确保它能生成正确的 XML 输出。建议首先编写能够处理“除了表达式之外的所有内容”的编译引擎，然后用无表达式的 *Square Dance* 程序来测试该引擎。接着，将其扩展成具有表达式处理功能的语法分析器，然后用 *Square Dance* 和 *Array Test* 程序来测试该语法分析器。

测试语法分析器

- 用你的 `CompilationEngine` 处理提供的测试程序，然后利用提供的 `TextComparer` 工具来比较生成的输出文件和提供的 `.xml` 比较文件。
- 因为你的语法分析器生成的输出文件将会有与提供的比较文件相同的名称和扩展名，所以建议你将它们放在另外单独的路径下。
- 注意到 XML 输出文件的缩排方式只是为了文本的易读性。Web 浏览器和提供的 `TextComparer` 工具会忽略空格。

第 11 章　编译器 II：代码生成

Compiler II: Code Generation

The syntactic component of a grammar must specify, for each sentence, a deep structure that determines its semantic interpretation.
语法规范的语法成分必须为每个句子指定具有确定语义解释的深层结构。
——Noam Chomsky(1928 ~)，语言学家

大多数程序员认为编译器没什么特别的。但如果仔细思考一下，就会发现编译器所具有的能力近乎神奇，能将高级程序翻译成二进制代码。在本书中通过编写 *Jack*（一门简单但现代的基于对象的语言）编译器来揭开整个翻译过程的神秘面纱。跟 Java 和 C#的编译器一样，整个 Jack 编译器也是基于双层结构的：一个处于**后端**（*back-end*）的虚拟机（在第 7 至 8 章中开发过），和一个**前端**（*front-end*）模块，该模块是高级语言和 VM 语言之间的一座桥梁，它由**语法分析器**（在第 10 章中开发）和**代码生成器**（本章将讨论的主题）组成。

虽然编译器的前端模块在概念上被分为两个模块，但是它们通常被合并为一个单一程序，本章也会这么做。在第 10 章中构建了能“理解（分析）”Jack 源程序的语法分析器。本章把该分析器扩展成完整的编译器，它能将“已经被理解的”高级语言的程序转换成一系列等价的 VM 操作。该方法遵循了构建多数编译器时所采用的“**结构化分析-综合**（*modular analysis-synthesis*）”范式。

现代高级编程语言是十分丰富和强大的，它们可以定义和使用大量的数据抽象（比如对象和函数），也可以实现包含精巧流程控制语句的算法，还可以构建具有任意复杂度的数据结构。然而，这些程序所最终运行的目标平台则是很简单的。一般它们仅仅提供一组用于存储数据的寄存器，和用于处理数据的原始指令集。因此，将程序从高级语言翻译到低级语言是个有意义的难题。如果目标平台是虚拟机，那么事情就简单一点，但是高级语言的表示和虚拟机的表示之间的差距仍然很大。

本章的**背景知识**部分介绍了完成编译器的开发所必需的背景知识：管理符号表（symbol table）；表示变量、对象和数组并生成相应代码；将流程控制命令翻译为底层指令。**规范详述**部分描述如何将 Jack 程序的语义映射到 VM 平台及其语言上，**实现**部分介绍了 API，通过它来调用生成代码的模块。**项目**部分提供了一步步的操作指示和测试程序，用于完成编译器的构建工作。

通过学习这些内容你能有何收获呢？一般来说，没有上过正式编译课程的学生没有机会去亲自开发完整的编译器。因此，读者如果按照本书所说从零开始构建 Jack 编译器，就会以相对较小的努力而学到一门重要课程的知识（当然，除非进行正式的编译理论学习，否则所获取的编译理论知识仍然是有限的）。此外，在编译器的代码生成部分所使用的一些技术相当有技巧。看到这些技巧付诸实践，让人再一次感到惊异：人类的创造性能够把原始的转换机器打造成近乎魔法的神奇玩意儿。

11.1 背景知识 Background

程序本质上就是一系列操纵数据的操作。因此，将高级程序编译成低级语言主要涉及两个主要的问题：**数据的翻译**和**命令的翻译**。

完整的编译任务是指将高级程序最终翻译成二进制代码。既然我们介绍的是双层的编译器架构，那么本章要假定编译器是要生成 VM 代码。这里就不再具体讨论已在虚拟机层级（第 7 章和第 8 章）讨论过的底层问题了。

11.1.1 数据翻译 Data Translation

程序能操纵很多变量**类型**（*types*），包括整型和布尔型等简单类型，以及数组和对象等复杂类型。另一个需要讨论的重要内容是变量的**生命周期**和**作用域**（*scope*）——即局部变量、全局变量、参数、对象成员字段等等。

对于程序中遇到的每个变量，编译器必须将其映射到目标平台中适合描述其**类型**的等价表示上。此外，编译器必须管理变量的生命周期和作用域，这与它的**类型相关**。本小节描述编译器如何处理这些任务，首先介绍**符号表**（*symbol table*）的概念。

符号表（Symbol Table） 高级程序引入并操纵很多**标识符**（*identifiers*）。当编译器遇到标识符 xxx 时，它需要知道 xxx 代表什么。它是变量名、类名称，还是函数名？如果它是变量，那么 xxx 是某个对象的成员字段，还是某个函数的参数？变量的种类又是什么，整数、布尔数、字符或某种数据类型？在编译器能用目标语言表达 xxx 的语义之前，必须解决这些问题。另外，每当在源代码中遇到 xxx 时，所有以上这些问题都必须被回答（用于代码生成）。

显然有必要了解程序引入的所有标识符，并且记录每个标识符在源程序中代表的内容及其被映射到目标语言中的何种结构上。大多数编译器通过使用**符号表**（*symbol table*）抽象来保存这些信息。在源代码中第一次遇到一个新标识符时（比如在符号声明中），编译器将它的描述添加到表中。在代码的其他地方遇到标识符时，编译器在符号表中查找该标识符，然后得到关于它的所有必要信息。下面是符号表的典型例子：

符号表（假设来自某子程序）

名 称	类 型	种 类	#
nAccounts	int	static	0
id	int	field	0
name	string	field	1
balance	int	field	2
sum	int	argument	0
status	boolean	local	0

当翻译含有标识符的高级代码时，符号表就是编译器的“罗赛塔石碑（Rosetta stone[1]）”。例如语句 `balance=balance+sum`。使用符号表，编译器能将这条语句翻译成相应的代码，

[1] Rosetta Stone，刻有碑文的石碑，其中包含有对埃及象形文字进行解码所需的关键信息，是现代埃及古文物学的重要基础。1799 年，一支法国团队在埃及的罗塞塔镇附近发现了该石碑，由此而得名。石碑现存放在英国伦敦的大英博物馆。——审校者

该代码反映了下面的事实：balance 是当前对象中编号为 2 的成员字段，sum 是当前运行的子程序中编号为 0 的参数。编译的其他细节依赖于目标语言。

因为大多数语言允许不同的程序单元使用相同的标识符来表示完全不同的变量，所以基本的符号表抽象就变得稍微有点复杂。为了支持这种表达的自由性，每个标识符都内在地与**作用域**（scope，即标识符能被感知的程序区域）相关联。这些作用域可嵌套，因此编译器约定：当相同的标识符在内层和外层（可以理解为位于某函数体内部/外部或者是一个类的内部/外部等）都被定义时，处于内层标识符的作用域将屏蔽外层同名标识符的作用域。例如，如果语句 x++出现在某个 C 函数中，C 编译器首先确定标识符 x 是否被定义在当前函数中，如果是，那么就生成局部变量自增的代码，否则编译器会确定 x 是否在文件的全局范围内定义了，如果是，那么就生成全局变量自增的代码。这种作用域的嵌套深度是没有限制的，因为某些语言允许定义这样的变量：这些变量仅对其被定义的代码块而言是局部的。

因此，符号表除了保存所有标识符相关的信息之外，它还必须以某种方式记录标识符的作用域。解决此问题的典型数据结构是由**哈希表**（*hash table*）组成的链表，每个哈希表反映了一种作用域，该作用域被嵌套在链表中下一个哈希表所描述的作用域中。当编译器没有在与当前作用域相关的哈希表中找到需要的标识符时，它将会在链表中的下一个哈希表中继续查找，也就是按照从内层作用域到外层作用域的顺序来寻找标识符。因此，如果 x 在某个代码段（比如某个方法）中没有被定义，那么 x 可能被定义在当前段的宿主代码段中（比如某个类），等等。

变量处理 编译器所面对的基本挑战之一是，如何将源程序中声明的各种变量类型映射到目标平台的内存中去。首先，不同的变量**类型**需要不同大小的内存块，因此这种映射不是一对一的。其次，不同的变量有不同的生命周期。例如，应该在程序的整个运行期中保持每个静态变量的单一副本。相比之下，对于对象的所有实例变量（成员字段），类的每个对象实例都应该保存有其各自的副本，在清除对象时，应该收回其内存。同样，每次调用子程序时，必须为其局部变量和参数变量创建新的副本。

上面说的算是坏消息。好消息是，我们已经解决了所有这些困难。在双层编译器架构中，变量的内存分配任务交给了 VM 后端程序。在第 7 章和第 8 章中构建的虚拟机包含了

用于提供标准类型变量的存储机制。这些标准变量类型包括：**静态变量**、**局部变量**、**参数变量**以及对象的**成员字段**，它们都是大多数高级语言所需要的。通过使用全局堆栈和虚拟内存段，这些变量的所有分配和去配细节已经在 VM 层被处理了。

这种功能并不是很轻易就实现的。实际上，必须费很大的劲儿去构建 VM 的实现，它将全局堆栈和虚拟内存段映射到最终的硬件平台上。然而付出的努力是值得的：对于任何给定的语言 *L*，任何 *L*-VM 编译器现在都已经完全从复杂的底层内存管理中解脱出来了。编译器所需要做的唯一的事情就是将源程序中的变量映射到虚拟内存段上，然后用 VM 命令来表达操控这些变量的高级命令，这就变成相对简单的任务了。

数组处理 数组（array）几乎总是被存储在连续的内存单元组成的内存段中（多维数组将被转化为一维数组）。数组的名字通常被当作是指针，它向内存中存储该数组的内存段基地址。在一些语言（比如 Pascal）中，声明数组时，用于存储该数组的整个内存空间被一次性分配。而其他一些语言（比如 Java），数组的声明仅仅只会导致分配一个指针（或称引用），该指针指向数组内存段的基地址。只有在程序运行期间该数组被真正构造时，它才会被分配合适的内存段。这是从内存堆（heap）中经由**动态内存分配**（*dynamic memory allocation*）得到的内存段，而动态内存分配是通过调用操作系统提供的底层内存管理程序实现的。操作系统有典型的 `alloc(size)` 函数，它知道如何找到可用大小为 `size` 的内存块，并返回它的基地址给调用者。因此，当编译器编译诸如 `bar=new int[10]` 这样的高级语句时，它生成执行操作 `bar=alloc(10)` 的低级代码。该代码导致将数组内存块的基地址赋给 `bar`，这正是我们所期望的。图 11.1 给出了这个例子的截图。

下面来考虑编译器如何翻译语句 `bar[k]=19` 。因为符号 `bar` 指向数组的基地址，该语句也能使用 C 语言`*(bar+k)=19` 来表示，也就是，“将 19 存储到地址为 `bar+k` 的内存单元中。”为了实现这个操作，目标语言必须具有某种间接寻址机制。该机制不是直接将一个值存储到地址为 y 的内存位置中去，而是要能够先将地址为 y 的内存单元中的值当作指针，然后将值存到该指针指向的内存单元中去。不同的语言有不同的方法来执行这个指针运算，图 11.2 给出了两种可能性。

Java代码

```
...
void foo (int k) {
  int x, y;
  int[] bar; // 声明一个数组
  ...
  // 构造数组
  bar = new int[10];
  ...
  bar[k] = 19;
}
...
Main.foo(2); // 调用foo方法
...
```

（执行 bar[k]=19 命令后，RAM 的状态如图）

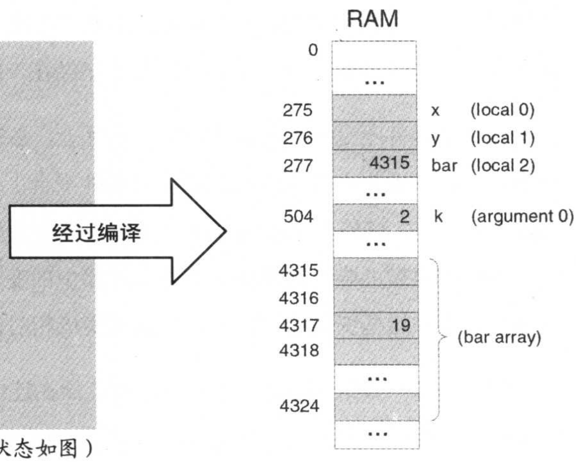


图 11.1 数组处理。因为内存的分配只有在程序运行期间才能确定，所以图中显示的地址是任意指定的

对象处理 某个类（比如，*Employee*）的对象实例封装了数据项（比如 *name* 和 *salary*），以及一组操纵它们的操作（operation，或称方法）。编译器对数据和操作的处理方式是完全不一样的。首先来看数据。

对象数据的底层处理方式与数组的处理方式很相似，也是在连续的内存段中存储每个对象实例。在大多数面向对象语言中，当一个“类”类型变量被声明时，编译器也仅分配一个指针变量。只有通过调用类构造函数来真正构建对象时，才给它分配所需的内存段。因此，当编译某个类 `Xxx` 的构造函数时，编译器首先利用类成员的数量和类型来决定在内存中需要多少个字（比如 *n*）来存储类 `Xxx` 的一个对象实例。接下来，编译器生成为新创建对象分配内存的代码，例如，`this=alloc(n)`。该操作将 `this` 指针设置为表示新创建对象所在内存段的基地址，正如我们所愿。图 11.3 以一段 Java 代码说明了这些操作。

伪 VM 代码

```
// bar[k]=19, 或 *(bar+k)=19
push bar
push k
add
// 使用指针访问 x[k]
pop addr // addr 指向 bar[k]
push 19
pop *addr // 将 bar[k] 设置为 19
```

最终的 VM 代码

```
// bar[k]=19, 或 *(bar+k)=19
push local 2
push argument 0
add
// 使用 that 虚拟段来访问 x[k]
pop pointer 1
push constant 19
pop that 0
```

图 11.2 数组处理。Hack 的 VM 代码(右边部分)遵循 7.2.6 小节介绍的惯例

因为每个对象被一个指向其基地址的指针变量所表示，所以可以使用相对于基地址的索引来访问对象所封装的数据。例如，假设 `Complex` 类包含下面的方法：

```
Public void mult (int c) {
  re = re * c;
  im = im * c;
}
```

编译器应该如何处理语句 `im = im * c` 呢？那么，编译器通过查看符号表就可以知道 `im` 是 `this` 对象的第二个成员，`c` 是 `mult` 方法的第一个参数。利用这些信息，编译器就能将 `im = im * c` 翻译成执行`*(this + 1) = *(this + 1) times (argument 0)`操作的代码。当然，生成的代码将会应用目标语言来实现该操作。

假设现在希望通过方法调用（比如 `b.mult(5)`）来调用 `mult` 方法操作对象 `b`。编译器应该如何来处理这个方法的调用呢？对于数据变量成员（如，`re` 和 `im`），每个对象实例都保存了**各自的副本**。然而对于类中的方法，它在目标代码层（target code level）只存在**唯一副本**，即**所有**该类的对象实例都共用同一个方法副本。为了使得每个对象实例看上去好像是封装了全部所需的代码，编译器必须保证这个唯一的方法副本能够对任意对象实例

Java 代码

```
class Complex {
  // 属性（字段）:
  int re; // 实部
  int im; // 虚部
  ...
  /** 构造新的 Complex 对象 */
  public Complex(int aRe, int aIm) {
    re = aRe;
    im = aIm;
  }
  ...
}
  // 下列代码可以放在任何类当中
  public void bla() {
    Complex a, b, c;
    ...
    a = new Complex(5,17);
    b = new Complex(12,192);
    ...
    c = a; // 仅拷贝了引用 (reference)
    ...
  }
```

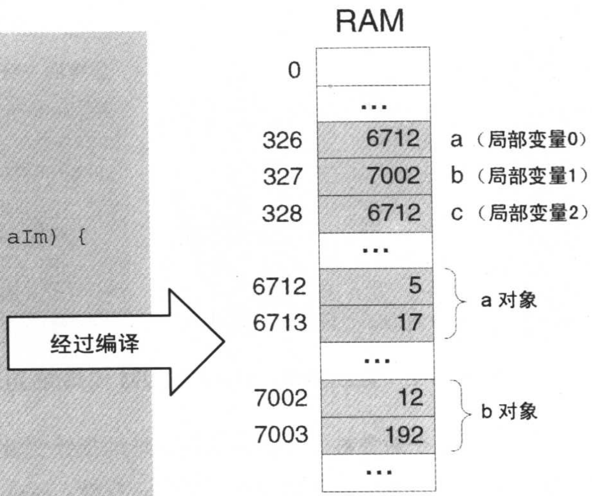


图 11.3 对象处理。因为内存的分配只有在程序运行期间才能确定，所以图中所示地址是任意指定的

进行方便的操作。这就需要将被调用方法的引用（reference，也称函数指针）作为隐含参数传递给对象实例，因此编译 `b.mult(5)`时，就好像它被写成了 `mult(b,5)`。一般来说，对方法的调用 `foo.bar(v1, v2,...)`被翻译成 VM 代码 `push foo`，`push v1`，`push v2`，……，`call bar`。这样，编译器就能使得同一个方法副本可以操作任何对象实例，并且形成“每个对象实例都封装了全部代码”的假象。

然而，编译器的工作并没有完成。因为语言允许不同类中的不同方法具有相同的名字，所以编译器必须确保对应的方法被应用在对应的对象上。而且，因为在子类中可能出现方法重载（method overriding），面向对象语言的编译器必须在程序运行期间来确定。如果不

支持以上操作（例如 Jack 语言），那就在编译期间来确定。对于如 `x.m(y)` 的方法调用，编译器必须保证调用的方法 `m()` 属于对象实例 `x` 所属的那个类。

11.1.2 命令翻译
Commands Translation

现在来描述高级命令是如何被翻译成目标语言的。因为已经讨论了变量、对象和数组的处理，所以现在只有两个问题需要讨论：**表达式求值**（*expression evaluation*）和**程序流程控制**（*control flow*）。

表达式求值 应该如何生成用于计算形如 `x+g(2,y,-z)*5` 的高级表达式代码呢？首先必须“理解”表达式的语法结构，比如将其转换成如图 11.4 所示的语法分析树。这个分析工作已经在第 10 章介绍的**语法分析器**中完成了。接下来可以在遍历这棵语法分析树的过程中生成对应的 VM 代码。

如何选择代码生成的算法，主要依赖于目标语言。对基于堆栈的目标平台而言，只需以**后缀**表示法（也称为 RPN，*Right Polish Notation* 即逆波兰表示法）来列印这棵树。

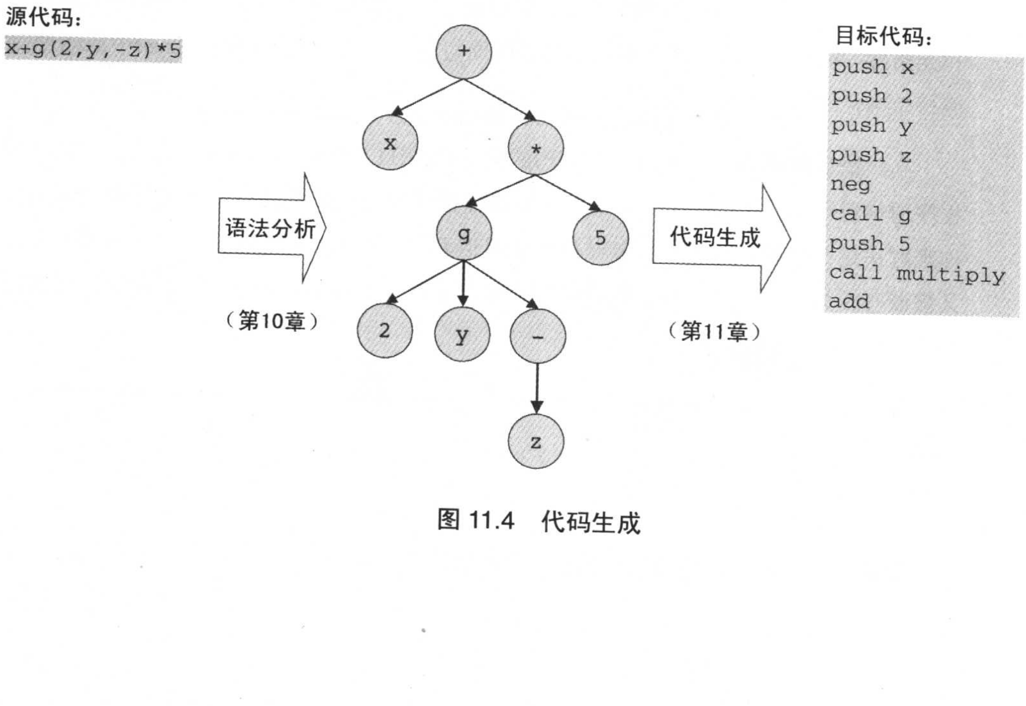


图 11.4 代码生成

在 RPN 语法中，诸如 $f(x,y)$ 的函数操作被表示成 x，y，f（在 VM 语言语法中表示为：`push x`，`push y`，`call f`）。同样的，诸如 $x+y$ 的操作，在前缀表示法中是 $+(x, y)$，在这里被表示成 x，y，$+$（也就是 `push x`，`push y`，`add`）。所以将表达式翻译成基于堆栈的 VM 代码的方法是很简单的，它基于对语法分析树递归的后序遍历，如下所示：

```
CodeWrite(exp):
if exp 是数字 n              then 输出 "push n"
if exp 是变量 v              then 输出 "push v"
if exp = (exp1 op exp2)      then CodeWrite(exp1), codeWrite(exp2),
                                  输出 "op"
if exp = op(exp1)            then CodeWrite(exp1), 输出 "op"
if exp = f(exp1 . . . expN)  then CodeWrite(exp1), . . . , CodeWrite(expN),
                                  输出 "call f"
```

该算法被应用在图 11.4 中的树，读者就可以验证其正确性，该算法生成的堆栈机代码如图 11.4 右边部分所示。

程序流程控制表达式的编译 高级编程语言都具有多种流程控制结构，比如 `if`、`while`、`for`、`switch` 等等。相比之下，低级语言一般提供两种基本的控制结构：**条件** *goto* 和**无条件** *goto*。因此，编译器的开发者所面对的挑战之一就是只使用这两个原始控制结构将结构化的代码段翻译成目标代码。如图 11.5 所示，翻译逻辑是相当简单的。

高级语言的两个特性使得控制结构的编译比图 11.5 中显示的要稍微复杂一点。首先，程序通常包含多个 `if` 和 `while` 语句。编译器能够通过生成并使用不同的标签（label）名称来解决这个问题。其次，控制结构可能是嵌套的，比如 `if` 被嵌套在 `while` 中，`while` 又被嵌套在另一个 `while` 中，等等。编译器可以通过使用递归编译方法来轻松解决这种复杂性。

11.2 规范详述 Specification

用法 Jack 编译器接受单一的命令行参数，如下所示：

```
promt> JackCompiler source
```

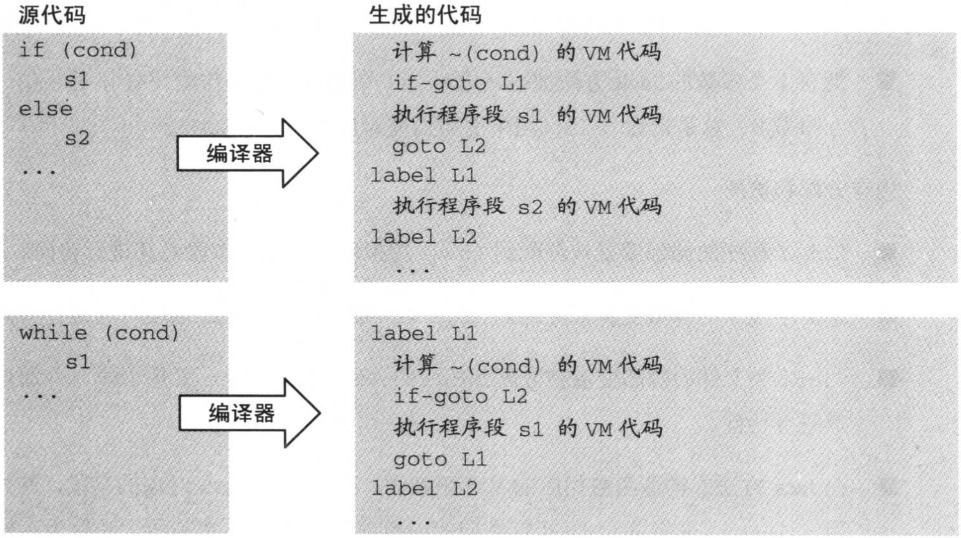


图 11.5 控制结构的编译

source 是形如 Xxx.jack（该扩展名是必要的）的文件名称或是包含若干个.jack 文件的路径名（在这种情况下没有扩展名）。编译器将各 Xxx.jack 文件编译成名为 Xxx.vm 的文件，并保存在与源文件相同的路径下。如果 *source* 是路径名，那么该路径下的每个.jack 文件都将被编译，并在相同路径下生成对应的.vm 文件

11.2.1 虚拟机平台之上的标准映射
Standard Mapping over the Virtual Machine

编译器将每个.jack 文件翻译成对应的.vm 文件，对于.jack 文件中的每个构造函数、函数、方法，该.vm 文件包含了相对应的 VM 函数（如图 7.8 所示）。如此一来，每个 Jack 编译器必须遵循下面的代码生成约定。

文件命名和函数命名 每个.jack 文件被编译成独立的.vm 文件。Jack 子程序（函数，方法，和构造函数）被编译成下列的 VM 函数：

- 在 Jack 程序中，类 `Yyy` 中的 Jack 子程序 `Xxx()` 被编译成名为 `Yyy.xxx` 的 VM 函数。
- 拥有 *k* 个参数的 Jack **函数**或**构造函数**被编译成对 *k* 个参数进行操作的 VM 函数。
- 拥有 *k* 个参数的 Jack 方法被编译成对 *k*+1 个参数进行操作的 VM 函数。第一个参数（参数 0）总是指针 `this`（用来引用当前对象）。

内存分配和访问

- Jack 子程序的**局部变量**被分配到 `local` 虚拟段，并通过该段对其进行访问。
- Jack 子程序的**参数变量**被分配到 `argument` 虚拟段，并通过该段对其进行访问。
- `.jack` 类文件的**静态变量**被分配到相应的 `.vm` 文件的 `static` 虚拟段，并通过该段对其进行访问。
- 在 Jack **方法**或**构造函数**相应的 VM 函数中，要想访问 `this` 对象的字段，首先通过使用 `pointer0` 来将以 `this` 指针指向的虚拟内存段（`this` 虚拟段）映射为存储当前对象的内存段上（实际就是使 `this` 指针指向存储当前对象内存段的基地址），然后通过 `this` *index* 来访问对象中的各个成员，这里 *index* 是非负数。
- 在 VM 函数中，要想访问数组的数据项，同访问对象的方法相似，首先通过使用 `pointer1` 使得 `that` 指针指向数组的第一个元素，然后通过 `that` 指针来访问数组中的数据项

子程序调用

- 在调用 VM 函数之前，调用者（本身也是 VM 函数）必须将函数的参数压入堆栈。如果被调用的 VM 函数是 Jack 方法，那么首先压入的参数必须是该方法操作对象的引用（reference）。
- 将 Jack **方法**编译成 VM 函数时，编译器必须插入适当的 VM 代码来设置 `this` 虚拟段的基地址。同样，在编译 Jack **构造函数**时，编译器也必须插入 VM 代码，该代码为新对象分配一个内存段然后将 `this` 指向这个内存段的基地址。

从返回类型为 Void 的方法及函数返回 高级 void 子程序并不返回值。这个抽象按如下方式来处理：

- 返回类型为 *void* 的方法和函数所对应的 VM 函数必须返回常数 0 作为其返回值。
- 编译 `do sub` 语句时（`sub` 是返回类型为 void 的方法或函数），对应的调用者必须弹出（并忽略）该返回值 （该值总是常数 0）。

常数

- `null` 和 `false` 被映射到常数 0。`true` 被映射到常数－1（该常数可以通过 `push constant 1`，再对其进行 `neg` 操作来得到）。

使用操作系统服务 基本的 Jack OS 是以一组名为 `Math.vm`、`Array.vm`、`Output.vm`、`Screen.vm`、`Keyboard.vm`、`Memory.vm` 和 `Sys.vm` 的 VM 文件（第 9 章中给出了这些文件的 API）来实现的。所有这些文件必须与编译器生成的 VM 文件保存在一起。因此，任何 VM 函数可以调用 OS 的任何 VM 函数。编译器在生成代码的时候会用到以下 OS 提供的函数：

- 乘法和除法由 OS 函数 `Math.multiply()` 和 `Math.divide()` 来处理。
- 字符串常数由 OS 构造函数 `String.new(length)` 来创建。字符串赋值如 `x="cc...c"` 可以通过对 OS 程序 `String.appendchar(nextChar)` 的一系列调用来实现。
- 构造函数利用 OS 函数 `Memory.alloc(size)` 来为新对象分配空间。

11.2.2 编译过程举例
Compilation Example

编译 Jack 程序（若干个 `.jack` 类文件）包含两个主要的任务：（1）使用前一章开发的编译引擎来分析代码；（2）根据之前给定的规范和原则来生成代码。图 11.6 给出了涉及本章诸多知识点的具体例子。

高级代码（BankAccount.jack 类文件）

```
/* 为了使范例的内容充分并保持简洁易懂，本例当中省略了一些银行业务的细节 */
   class BankAccount {
   // 类中的成员变量
   static int nAccounts;
   static int bankCommission; // 百分比，比如 10 代表 10%
   // 账户的属性
   field int id;
   field String owner;
   field int balance;

   method int commission(int x) { /* 这里省略了实现代码 */ }

   method void transfer(int sum, BankAccount from, Date when) {
   var int i, j; // 一些局部变量
   var Date due; // Date 是用户自定义类型
   let balance = (balance + sum) - commission(sum * 5);
   // 其他代码...
   return;

   }
   // 其他的方法...
}
```

类作用域（class-scope）内的符号表

名称	类型	种类	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

方法作用域（method-scope）内的符号表

名称	类型	种类	#
this	BankAccount	argument	0
sum	Int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

图 11.6 代码生成范例，集中描述如下语句的翻译过程：

let balance = (balance+sum)-commission(sum*5)

伪 VM 代码

```
function BankAccount.commission
  // 这里省略了部分代码
function BankAccount.transfer
  // 将 "this" 指向被传入的对象
  // (省略了部分代码)
  push balance
  push sum
  add
  push this
  push sum
  push 5
  call multiply
  call commission
  sub
  pop balance
  // 更多代码
  push 0
  return
```

最终的 VM 代码

```
function BankAccount.commission 0
  // 这里省略了代码
function BankAccount.transfer 3
  push argument 0
  pop pointer 0
  push this 2
  push argument 1
  add
  push argument 0
  push argument 1
  push constant 5
  call Math.multiply 2
  call BankAccount.commission 2
  sub
  pop this 2
  // 更多代码...
  push 0
  return
```

续图 11.6 代码生成范例，集中描述如下语句的翻译过程：

`let balance = (balance+sum)-commission(sum*5)`

11.3 实现
Implementation

现在要为整个编译器打造软件架构。该架构建立于第 10 章所描述的**语法分析器**之基础上。事实上，当前架构的基础正是逐步将语法分析器发展成完整编译器的过程。因此整个编译器由 5 个模块构成：

- `JackCompiler`： 建立和调用其他模块的顶层驱动模块；
- `JackTokenizer`： 字元转换器；
- `SymbolTable`： 符号表；
- `VMWriter`： 生成 VM 代码的输出模块；
- `CompilationEngine`： 自顶向下的递归式编译引擎。

11.3.1 *JackCompiler* 模块
The *JackCompiler* Module

编译器对形如 `Xxx.jack` 的文件名称（或是包含若干`.jack` 文件的路径名）进行编译操作。对于每个 `Xxx.jack` 输入文件，编译器创建一个 *JackTokenizer* 和 `Xxx.vm` 输出文件。接着，编译器利用 *CompilationEngine*、*SymbolTable* 和 *VMWriter* 模块来进行文件输出。

11.3.2 *JackTokenizer* 模块
The *Jack Tokenizer* Module

字元转换器的 API 已经在 10.3.2 小节给出。

11.3.3 *SymbolTabel* 模块
The *SymbolTabel* Module

该模块提供创建和使用**符号表**（symbol table）的服务。前面提过每个符号都具有作用域，符号只有在源代码中的这个作用域内才是可见的。符号表通过为每个在作用域里的符号赋予一个动态变化的数值（索引 index）来实现。索引从 0 开始，每当一个标识符被添加，就自增 1，当开始新作用域时，索引就被重置为 0。符号表中可以有下列标识符：

static: 作用域：类。
field: 作用域：类。
argument: 作用域：子程序（方法/函数/构造函数）。
var: 作用域：子程序（方法/函数/构造函数）。

若编译的是正确合法的 Jack 代码，则任何没有在符号表中找到的标识符就被认为是子程序名或类名称。因为 Jack 语言语法规则能够区分这两种可能性，鉴于编译器也没有必要做这种“连接”，因此没有必要把这些标识符保存在符号表中。

SymbolTable：符号表的抽象描述。符号表将程序中的标识符名称与其在编译过程中所需属性（类型、分类、索引）关联起来。Jack 程序的符号表有两种嵌套范围（类/子程序）。

程 序	参 数	返回值	功 能
Constructor	—	—	创建新的空符号表
startSubroutine	—	—	开创新的子程序作用域 （即将子程序的符号表重置）
Define	name(String) type(String) kind(STATIC, FIELD,ARG, 或 VAR)	—	定义给定了**名称**、**类型**和**分类**的新标识符，并赋给它一个索引。STATIC 和 FIELD 标识符的作用域是整个类，ARG 和 VAR 的作用域是整个子程序
VarCount	Kind(STATIC, FIELD,ARG, 或 VAR)	Int	返回已经定义在当前作用域内的变量的数量
KindOf	name(String)	(STATIC, FIELD, ARG, VAR, NONE)	返回当前作用域内的标识符的**种类**。如果该标识符在当前作用域内是未知的，那么返回 NONE
TypeOf	Name(String)	String	返回当前作用域内标识符的**类型**
IndexOf	name(String)	int	返回标识符的**索引**

实现提示 符号表抽象和 API 能够利用两个独立的哈希表来实现：一个用于类的作用域，另一个用于子程序作用域。当启动新的子程序时，原来的子程序作用域表就被清空。

11.3.4 *VMWriter* 模块
The *VMWriter* Module

VMWriter：使用 VM 命令语法，将 VM 命令写入文件。

程 序	参 数	返回值	功 能
Constructor	输出文件/流	—	创建新的待写文件
writePush	segment(CONST, ARG, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) index(int)	—	写入 VM push 命令
writePop	segment(CONST, ARG, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) index(int)	—	写入 VM pop 命令
writeArithmetic	command(ADD, SUB, NEG, EQ, GT, LT, AND, OR, NOT)	—	写入 VM 算术命令
writeLabel	label(String)	—	写入 VM label 命令
writeGoto	label(String)	—	写入 VM goto 命令
writeIf	label(String)	—	写入 VM If-goto 命令
writeCall	name(String) nArgs(int)	—	写入 VM call 命令
writeFunction	name(String) nArgs(int)	—	写入 VM function 命令
writeReturn	----	—	写入 VM return 命令
close		—	关闭输出文件

11.3.5 *CompilationEngine* 模块 The *CompilationEngine* Module

该类自行编译。它从 `JackTokenizer` 读取输入，然后将输出写入 `VMWriter` 。该模块以一系列 `compilexxx()`程序组织起来，xxx 是 Jack 语言的语法元素。这些子程序之间的约定是：每个 `compilexxx()`程序应该从输入中读取语法元素 xxx ，再调用 `advance()` 得到输入的下一个语法元素 xxx ，然后将 xxx 语义对应的 VM 代码进行输出。因此仅当输入中的下一个语法元素是 xxx 时，对应的 `compilexxx()`才会被调用。如果 xxx 是表达式的一部分，它就具有一个值，输出的代码应计算结果并且将结果置于 VM 的堆栈顶。

该模块的 API 与第 10 章中语法分析器的 *CompilationEngine* 模块的 API 相同，因此建议逐步将语法分析器演化成完整的编译器。11.5 节提供了逐步的操作指示以及相应的测试程序。

11.4 观点 Perspective

Jack 是一门相对简单的编程语言，这个事实回避了几个让人头疼的编译问题。例如，Jack 看上去像是一种带类型的语言（typed language），然而事实并非如此。所有的 Jack 数据类型都是 16-位长的，语言的语义允许 Jack 编译器忽略几乎所有的类型信息。因而，当编译和计算表达式时，Jack 编译器不需要确定它们的类型（有一个例外：编译方法调用 `x.m()`时，需要确定 x 的“类”类型）。同样的，Jack 中的数组数据项也没有被指定类型。相比之下，大多数编程语言都具有丰富的类型系统，并对其编译器产生重要影响：对于不同类型的变量必须分配不同数量的内存；从一种类型到另一种类型的转换需要特定的语言操作；对诸如 `x+y` 的简单表达式的操作，很大程度上依赖于 x 和 y 的类型；等等。

另一个重大简化就是 Jack 语言没有支持继承（inheritance）。这意味着可以在编译期静态地处理所有方法调用。相比之下，支持继承的语言编译器必须将方法作为虚拟方法对待，并根据对象的运行期类型来确定它们的具体位置。例如，考虑方法调用 `x.m()`。如果语言支持继承，x 就有可能属于多个类，这就无法在运行期之前确定它具体属于哪个类。因此，如果在 x 所属的类中找不到方法 m 的定义，还可能在该类的父类中找到。

Jack 所不支持的另一种面向对象的重要属性是公共类成员字段（public class fields）。例如，若 `circ` 是类型为 `Circle` 的对象，该类有属性 `radius` ，我们就不能通过编写诸如 `r=circ.radius` 这样的语句来访问 `circ` 对象中的 `radius` 成员，为此，必须为访问 `Circle` 类中的成员编写并调用专门的读取方法：`r=circ.getRaius()`来访问 `radius` 成员（这是优秀的编程实践）。

因为缺少对类型、继承和公共成员的支持，所以各个类在编译过程中可以独立地进行编译而无需考虑其他类成员。绝对不能对一个类的成员进行直接访问，所有与其他类中方法的联系都仅仅通过名称来处理。

Jack 语言许多其他简陋的特性不是很突出，而且可以很容易地来化解这些简陋所带来的麻烦。例如，可以用 `for` 和 `switch` 语句来扩展语言设施。同样，目前 Jack 语言还不支持将常数值（比如`'c'`）赋给 `char` 类型变量的能力，但完全可以添加这个功能。（为了将常量`'c'`赋给 Jack `char` 变量 `x`，必须首先将`"c"`赋给 `String` 变量（比如 `s`），然后使用 `let x=s.charAt(0)`。显然，如果能像在 Java 语言中那样，简单地用语句 `let x='c'`表示，那当然是很好）。

最后要说的是，一如往常，本书没有关心优化问题。考虑一下“`c++`”这条高级语句。简陋的编译器会将其翻译成一系列低级 VM 操作 `push c`，`push 1`，`add`，`pop c`。接下来，VM 实现会将每个 VM 命令翻译成一系列机器指令，生成相当大的代码段。而优化过的编译器会注意到我们仅在处理简单的自增操作，它会将其翻译成简单的机器指令，比如翻译成 Hack 平台上的两条机器指令`@c` 和 `M=M+1` 。当然，这还只是工业级强度编译器需要具备的特性的例子之一，还有许多其他要求。总之，时间和空间效率在编译器的代码生成部分和编译过程中起着很重要的作用。

11.5 项目 Project

目标 将第 10 章中构建的**语法分析器**扩展成完整的 Jack 编译器。要逐步用“生成可执行 VM 代码”的模块来取代“生成 XML 代码”的模块。

资源 本项目中的主要工具是用于实现编译器的编程语言。同时还需要 Jack 操作系统的可执行副本和 VM 仿真器，用提供的一组测试程序来测试编译器生成的代码。

约定 完成 Jack 编译器的实现。编译器输出的 VM 代码应该可以在项目 7 和项目 8 所构建的虚拟机上运行。利用你的编译器来编译这里给出的所有 Jack 程序。确认编译过的程序执行了正确的操作。

阶段 1：符号表

建议你首先构建编译器的符号表模块，再利用它来扩展项目 10 中构建的语法分析器（syntax analyzer）。目前，任何时候在程序中遇到标识符（identifier）时（比如遇到 `foo` 这样一个标识符），语法分析器将输出 XML：`<identifier> foo </identifier>` 。现在换一种方案，让语法分析器输出下列信息，作为其 XML 输出的一部分（使用任何你决定采用的格式）：

- 标识符的分类（*var*，*argument*，*static*，*field*，*class*，*subroutine*）；
- 标识符当前是否已经被**定义**（比如，该标识符代表在 `var` 语句中被声明的变量）或已经被**使用**（比如，该标识符代表表达式中的变量）；
- 标识符是否表示属于四种分类（*var*，*argument*，*static*，*field*）之一的变量，符号表是否给标识符赋予了索引号。

可以通过项目 10 中提供的 Jack 测试程序来运行经过扩展的语法分析器，以此来测试你的符号表模块和前面所述的功能。一旦你的（经过扩展的）语法分析器的输出包含了上述这些信息，这就意味着你已经赋予语法分析器理解 Jack 程序语义的能力。本阶段你可以得到完整的编译器，并用 VM 代码输出来取代原来的 XML 输出。可以通过逐步扩展语法分析器的代码来实现完整的编译器。

阶段 2：代码生成

关于如何开发编译器的代码生成功能，我们不再详述特定的指导方案，贯穿本章的范例已经给出了很多提示。本书提供一组 6 个程序来逐步地对各种特性进行单元测试。强烈建议按这些给定的顺序来测试编译器。这样，根据每个测试程序的要求，你将会了解如何分阶段来构建编译器的代码生成机制。

操作系统 Jack OS（第 12 章的主题）是用 Jack 语言编写的。OS 源代码（通过完整无误的 Jack 编译器）被翻译成一组 VM 文件，这组文件就是 *Jack OS*。想要在 VM 仿真器上运行应用程序，不仅要加载应用程序的`.vm`文件，还要加载 OS 的所有`.vm`文件。于是，当应用级的 VM 函数调用某个 OS 级的 VM 函数时，它们会在相同的环境下找到对方。

测试方法 通常，编译程序并遇到问题时，你会推断一定是程序有些毛病，然后调试它。在本项目中，情况就完全相反了。本书提供的所有测试程序都是没有错误的。因此，如果在它们的编译过程中出现了错误，那么需要修改的就是你的**编译器**，而不是测试程序。对于每个测试程序，推荐按下列步骤来测试：

1. 将`tools/os`路径下的所有提供的 OS `.vm`文件，以及包含测试程序的`.jack`文件拷贝到程序的路径下。

2. 利用你的编译器来编译程序路径。这个操作应该仅仅编译路径下的`.jack`文件。

3. 如果出现了编译错误，请修改你的编译器然后返回到步骤 2（注意：所有提供的测试程序都是没有错误的）。

4. 在这里，程序路径应该包含对应于每个源`.jack`文件的`.vm`文件，以及所有提供的 OS `.vm`文件。若非如此，请修改你的编译器然后返回到步骤 2。

5. 在 VM 仿真器中执行翻译后的 VM 程序，加载整个路径，然后使用“no animation”模式。6 个测试程序都包含详细的执行细节，在下面会介绍。

6. 如果程序没有按照预期的方式执行，或者在 VM 仿真器中显示了某个错误消息，请修改你的编译器然后返回到步骤 2。

测试程序

在这里我们提供 6 个测试程序。所有程序都是用来逐步地对你的编译器的语言处理能力进行单元测试。

Seven 该程序计算`(3*2)+1`的值，然后在屏幕的左上角打印结果。为了测试你的编译器是否正确地编译了程序，将编译后的代码放在 VM 仿真器中运行，确认它正确地显示了 7（正确结果）。

目的：测试你的编译器如何处理简单程序，该程序包含了带有整数常数（不带变量）的运算表达式、一个`do`语句和一个`return`语句。

十进制-二进制转换 该程序将 16-位十进制数转换成它的二进制表示形式。该程序从`RAM[8000]`读取一个十进制数，将其转换成二进制数，然后将这 16 个独立的位存储在`RAM[8001…8016]`中（每个位置包含 0 或 1）。在转换开始之前，程序将`RAM[8001…8016]`初始化为－1。为了测试你的编译器是否正确地编译了程序，将编译后的代码放在 VM 仿真器中运行，然后按下列步骤进行：

- 在交互模式下将一个 16-位十进制数放入`RAM[8000]`中。
- 运行程序几秒钟，然后停止执行。
- 在交互模式下检查`RAM[8001…8016]`中是否包含了正确的结果。

目的：测试你的编译器是否正确地处理了 Jack 语言的所有过程化元素，即**表达式**（不带有数组或函数调用）、**函数**和所有**语句**。该程序没有测试对方法、构造函数、数组、字符串、静态变量和字段变量的处理能力。

Square Dance 该程序是一个小型的交互式“游戏”，它允许游戏者利用键盘上的四个方向键在屏幕上移动一个黑色方块。在移动过程中，方块的尺寸可以通过按“z”和“x”键来增加和减少。要退出游戏，就按“q”键。为了测试你的编译器是否正确地翻译了程序，将翻译后的代码放在 VM 仿真器中运行，确认它的行为符合以上描述。

目的：测试你的编译器是如何处理 Jack 语言的面向对象结构：**构造函数**、**方法**、成员**字段**和包含了**方法调用**的表达式。该程序没有测试对静态变量的处理能力。

Average 该程序计算一组用户提供的整数的平均数。为了测试你的编译器是否正确地翻译了程序，将翻译后的代码放在 VM 仿真器中运行，然后按照屏幕上显示的指示来操作。

目的：测试你的编译器如何处理**数组**和**字符串**。

Pong 一个球随机地在屏幕上移动，遇到屏幕的“墙壁”就反弹。游戏者可以通过按下键盘的左和右箭头键来水平地移动一个小拍子。拍子每击球一次，游戏者会得一分，同时拍子也会缩短一点来使得游戏更难。如果游戏者没有接到球，球会落到屏幕的底部水平线，游戏结束。为了测试你的编译器是否正确地翻译了程序，将翻译后的代码放在 VM 仿真器中运行，然后开始玩游戏（至少要得分，以便测试程序中在屏幕上显示分数的程序）。

目的：测试你的编译器如何处理**对象**（包括**静态变量**）。

Complex Arrays 利用数组执行 5 个复数计算。对于每次计算，程序在屏幕上打印期望结果和实际结果（由编译后的程序计算所得）。为了测试你的编译器是否正确地翻译了程序，将翻译后的代码放在 VM 仿真器中运行，确认实际结果与期望结果是相同的。

目的：测试你的编译器如何处理复杂的**数组引用**（*array reference*）和**表达式**。

第 12 章 操作系统

Operating System

Civilization progresses by extending the number of operations that we can perform without thinking about them.

对于不用经过特意思考就能执行的操作，其数量的增长推动文明进步。

——Alfred North Whitehead （1861 ~ 1947），数学家，

Introduction to Mathematics（1911）

本书前面的章节描述并构建了称为 *Hack* 的计算机平台的硬件架构，以及该硬件架构得以运作的软件层级。我们介绍了基于对象的语言 *Jack*，并且描述了如何为 Jack 编写编译器。其他高级编程语言也能在 Hack 平台上应用，只是每一门语言都需要各自的编译器。

在整个构建中的最后一个接口就是**操作系统**（OS）。OS 的作用就是来衔接计算机的硬件系统和软件系统，以使得整个计算机对程序员和用户而言更容易使用。例如，为了使得文本“`Hello World`”在计算机屏幕上显示，必须在特定的屏幕位置上画几百个像素。这可以通过参考硬件规范，编写相关的代码来完成，该代码在驻留 RAM 的屏幕映像中放置必要的比特位。显然，高级程序员希望事情能够变得更好些。他们希望使用诸如 `printString("Hello World")`的命令，然后让别人来负责其中的实现细节。操作系统在其中就起到了很重要的作用。

贯穿本章，**操作系统**这个术语使用比较宽泛。实际上，本书所描述的 OS 服务包含了最小规模的操作系统，主要是：（1）以一种对软件友好的方式封装了不同的硬件服务；（2）用不同的函数和抽象数据类型扩展了高级语言。在这个意义上的操作系统与语言的标准程序库的分界线就不是那么明显。事实上，某些现代语言（例如 Java）就趋向于将很多经典的操作系统服务（比如 GUI 管理，内存管理，和多任务处理等）连同很多语言扩展一起打包到其标准程序库中。

按照这种模式，本章描述并构建的服务就可以被看作是简单的 OS 和 Jack 语言的标准程序库的集合。该 OS 被打包成一组 Jack 类，每个类通过 Jack 子程序调用来提供一组相关的服务。这个 OS 与工业级强度的操作系统有很多相似的特点，但是它仍然缺少大量的 OS 特性，比如进程管理、磁盘管理、通信等。

操作系统通常是由高级语言编写，并被编译成二进制形式，就像任何其他程序一样。Jack 的 OS 也不例外，可以完全由 Jack 编写而成。然而与其他由高级语言编写的程序不同的是，操作系统代码必须了解它所运行的硬件平台。换句话说，为了对高层的应用程序员隐藏这些硬件细节，OS 程序员必须编写能够直接操纵这些细节的代码（这个任务需要参考硬件规范文档）。方便的是，这可以利用 Jack 语言来做到。正如本章将看到的，Jack 语言被定义成足够“低层级的”语言，以便能够允许它在需要时可以与硬件进行“亲密接触”。

本章的**背景知识**部分相对较长，描述了通常被用来实现基本操作系统服务的关键算法。这些服务包括数学函数、字符串操作、内存管理、文本和图形输出到屏幕的处理，以及从键盘输入的处理。**规范详述**部分，提供了 Jack OS 的完整的 API。**实现**部分描述了如何使用前面介绍的算法来构建 OS。最后的**项目**部分提供了所有必须的项目资源，用来逐步构建并对本章介绍的整个 OS 进行单元测试。

本章介绍了两个关键知识领域，一是涉及软件工程，二是涉及计算机科学。首先，完成了高级语言、编译器以及操作系统三个部分的构建。其次，因为操作系统的服务必须有效地执行，因此我们着重考虑运行期间的效率问题。由此将介绍一系列优秀算法，都是计算机科学领域的宝贵财富。

12.1 背景知识 Background

12.1.1 数学操作 Mathematical Operations

计算机系统必须支持诸如加法、乘法和除法这样的数学操作。通常，加法在 ALU 级的硬件中实现（在第 3 章中介绍过）。其他操作如乘法和除法，可以根据计算机的性价比

要求，通过硬件或软件来处理。这一小节介绍乘法、除法以及平方根操作是如何在在 OS 级的软件中被有效实现的。要注意的是，这些数学操作的硬件实现同样也基于这里提到的算法。

效率第一 数学算法操作在 n-位数上，在典型的计算机体系中，n=16, 32 或 64。通常，期望运行时间与参数 n 成比例（或者至少是 n 的多项式）的算法。运行时间与 n-比特位的**数值**成比例的算法是不可接受的，因为这些值是以 n 为指数的。例如，假设利用反复的加法算法来实现乘法操作 $x \cdot y$，则有 *for* $i = 1 \ldots y$ $\{result = result + x\}$。现在问题是在 64-位计算机中，y 可以比 18 000 000 000 000 000 000 更大，意味着这个幼稚的算法即便是在最快的计算机上都要运行好几年。相比之下，下面介绍的乘法算法的运行时间就不是与被乘数的值（可以大到 2 的 n 次方）成比例，而是与 n 成比例。因此，对于任意的乘数与被乘数，该算法仅需要 $c \cdot n$ 个基本操作，这里 c 是个很小的常数，它代表在每个循环迭代中执行的基本操作的数量。

使用标准的“Big-Oh（大 O）”表示法，即 $O(n)$，来描述算法的运行时间。读者如果不熟悉这种表示法，可以把 $O(n)$当作“按照 n 的数量级顺序的一种表示”。有了这个概念，就可以开始介绍一种有效的 n-比特位的数的乘法 $x \cdot y$ 算法，它的运行时间是 $O(n)$，而不是 $O(x)$或 $O(y)$。

乘法 回忆一下小学教过的标准乘法。为了计算 356 乘以 27，我们将两个数排列起来，一个在另一个之上。接着，我们用 7 来乘上 356 的每个数字。然后，“向左移动”一个位置，用 2 来乘上 356 的每个数字。最后，我们把每一列的和加起来，于是得到最后的结果。这种方法的二进制版本（如图 12.1 所示）也是按照相同的逻辑来计算的。

图 12.1 中的算法对 n-比特位的数执行 $O(n)$次加法操作，这里 n 是 x 和 y 中的位数。注意图中 *shiftedX* * 2 这个操作可以通过两种方式实现：左移其二进制比特位；或通过 *shiftedX* 加上自身来实现。利用原始的 ALU 操作可以很容易地执行这两种操作。因此这个算法在软件和硬件实现中都有着独特的优势。

Long multiplication

x				1	0	1	1	=	11	
y		*			1	0	1	=	5	j-th bit of y
				1	0	1	1			1
			0	0	0	0				0
		1	0	1	1					1
$x \cdot y$	1	1	0	1	1	1		=	55	

```
multiply(x, y):
  // Where x, y ≥ 0
  sum = 0
  shiftedX = x
  for j = 0...(n − 1) do
     if (j-th bit of y) = 1 then
        sum = sum + shiftedX
     shiftedX = shiftedX * 2
```

图 12.1 两个 *n*-bit 位数的乘法

关于算法表示的说明 本章的算法都是使用伪代码表示的，具有良好的可续性。唯一不明显的规则就是，使用缩排来代表代码块（避免了花括号或者 begin/end 关键字）。例如，在图 12.1 中，*sum* = *sum* + *shiftedX* 属于 *if* 语句的单一语句体，然而 *shiftedX* = *shiftedX* * 2 结束了 *for* 语句的双语句体。

除法 计算两个 *n*-比特位的数的除法 *x*/*y*，一种简陋的方法是重复地从 *x* 上减去 *y*，直到无法继续（即直到 *x*<*y*）。该算法的运行时间明显与除法的商成正比，可能大到 $O(x)$，即以 *n* 为指数。为了提高该算法的效率，可以在每次迭代中试着从 *x* 上减去 *y* 的一个较大的倍数。例如，如果 *x*=891，*y*=5，这样马上就能断定可以从 *x* 上减去一百个 5，剩余数仍然比 5 大，与原算法相比因此减少了 100 次迭代。事实上，这正是学校所教的长除法（long devision）*x*/*y* 方法的基本原理。在每次迭代中，试图从 *x* 上减去 *y* 的最大可能的位移，也

```
divide (x, y):
  // Integer part of x/y, where x ≥ 0 and y > 0
  if y > x return 0
  q = divide(x, 2 * y)
  if (x − 2 * q * y) < y
     return 2 * q
  else
     return 2 * q + 1
```

图 12.2 除法

就是 $y \cdot T$，这里 T 是最大的以 10 为幂的数，并且满足 $y \cdot T \leq x$。这种算法的二进制版本也是一样，只不过这时 T 是以 2 为幂而不是 10。

按照上面的思路来编写算法是很容易的事情。但若把相同的逻辑用可能更容易实现的递归程序来表示，就更具启发意义（如图 12.2 所示）。

该递归算法的运行时间由递归的深度来决定。因为在每次递归中，y 的值被乘上 2，且一旦出现 $y > x$ 立即结束，于是递归深度受到 n（x 的位数）的限制。每个递归层都包含了一个常数数量的加法，减法和乘法操作，意味着整个运行时间为 $O(n)$个这样的操作。

因为每个乘法操作也需要 $O(n)$次加法和减法操作，所以该算法被认为是次优的。但如果仔细观察，会发现可以不使用乘法来计算乘积$2 \cdot q \cdot y$。我们可以根据在前一个递归层的该乘积值，使用加法来求出其当前的值。

平方根 可以采用很多不同的方法来有效地计算平方根，例如，牛顿-拉夫逊方法（Newton-Raphson method）或者泰勒级数展开式（Taylor series expansion）。对我们而言，采用较简单的方法就够了。平方根函数 $y = \sqrt{x}$ 有两个有利的特性。首先，它是单调递增的。其次，我们已经知道如何来计算其反函数 $x = y^2$（利用乘法进行计算）。综合考虑这两个特点，有理由使用**二叉查找**（*binary search*）来计算平方根。图 12.3 给出了具体细节。

要注意的是每个循环迭代都进行了一个常数数量的算术操作。因为迭代的数量受到 $n/2$ 的限制，所以算法的运行时间是 $O(n)$次算术操作。

```
sqrt(x):
  //计算 y=√x 的整数部分。策略:
  //通过在 0...2^(n/2) − 1 范围内执行二叉搜索，来
  //确定一个满足条件 y^2 ≤ x < (y+1)^2 (对于 0 ≤ x ≤ 2^n) 的 y
  y = 0
  for j = n/2 − 1 . . .0 do
    if (y+2^j)^2 ≤ x  then y = y+2^j
  return y
```

图 12.3 应用二叉搜索来计算平方根

12.1.2 数字的字符串表示 String Representation of Numbers

计算机在内部使用二进制码来表示数字。然而人们习惯于用十进制的表示法来处理数字。因此，当人们要读取或者输入数字时，计算机必须要执行二进制到十进制的转换或者十进制到二进制的转换。通常，这些工作是由操作系统提供的底层服务程序来处理。现在来介绍如何实现这些 OS 服务。

当然，我们关注的是字符的一个子集，即 10 个表示数字的字符。这 10 个字符的 ASCII 码如下所示：

字符：	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII 码：	48	49	50	51	52	53	54	55	56	57

通过上述对照表，可以发现单个数字字符可以很容易地转换成对应的 ASCII 码表示，反之亦然。为了计算给定数字 $0 \le x \le 9$ 的 ASCII 码，只需将 x 加上 48 即 ‘0’ 的 ASCII 码即可。同理，ASCII 码 $48 \le c \le 57$ 所表示的数字值可以通过 $c-48$ 来获得。一旦知道了如何转换这些数字，就能转换任意给定的整数了。这些转换算法可以基于迭代或递归算法，图 12.4 和图 12.5 中分别给出了对应的算法。

12.1.3 内存管理 Memory Management

动态内存分配 计算机的程序会声明并使用各种类型的变量，包括如整数、布尔数等简单的数据类型以及如数组、对象等复杂的数据类型。高级语言的最大优点之一是程序员

```
// 将非负数转换成字符串
int2String(n):
lastDigit = n % 10
c = 代表 lastDigit 的字符
if n < 10
return c (作为字符串返回)
else
return int2String(n/10).append(c)
```

```
// 将字符串转换成非负数
string2Int(s):
v = 0
for i = 1 . . . s 的长度 do
d = 数字 s[i]的整数值
v = v * 10 + d
return v
// 假设 s[1]是字符串 s 的第一个字符
```

图 12.4、图 12.5　字符串-数字转换

不必关心内存管理细节：比如为变量分配内存空间；以及当该变量不再使用时，回收为其分配的内存空间。所有关于内存管理的琐碎工作都由编译器、操作系统和虚拟机在后端来完成。这一小节将描述操作系统在内存管理中所担当的角色。

不同变量的内存在程序生命周期中的不同时刻被分配。例如，**静态变量**在编译期间由编译器为其分配内存，而**局部变量**则在每个子程序开始运行时被分配在堆栈内。其他变量的内存则是在程序的执行过程中被动态分配，这就需要操作系统的帮助了。例如，每当 Java 程序创建新数组或新对象时，对应的内存块就会被分配，而这块内存块的大小只有在程序运行期间才能确定。当数组或对象不再使用时，其内存空间会被收回。在一些高级语言中（如 C++和 Jack），释放不再使用的内存空间是程序员的任务，然而在其他语言如 Java 中，会自动地产生“垃圾回收”。被用于进行动态内存分配的内存段称为堆（heap），负责管理这个资源的就是操作系统。

操作系统使用不同的技术来处理动态内存分配和去配。这些技术通常在两个称为 `alloc()`和 `deAlloc()`的函数中实现。我们提供两个算法版本：基本的算法版本和改进的算法版本。

基本内存管理算法　该算法所管理的数据结构就是一个单一的指针，称为 *free* ，它指向还未经过分配的内存的堆基地址。图 12.6a 展示了该算法。

```
Initialization: free = heapBase

// 分配内存块，大小为 size 个字(word)
alloc(size):
pointer = free
free = free + size
return pointer

// 去配指定对象所占用的内存空间
deAlloc(object):
do nothing
```

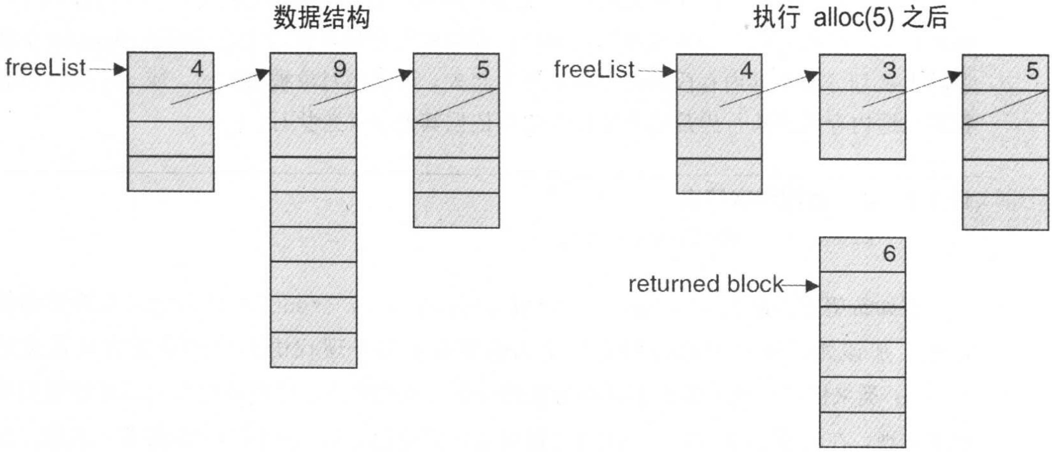
图 12.6a 基本内存分配方案（浪费较严重）

显然，这个算法对内存的浪费程度很严重，因为它不回收那些不再被使用的变量所占用的内存。

改进的内存分配算法 该算法管理一个由可使用内存段构成的链表，称为 *freeList*。每个内存段包含两个字段：该内存段的长度和指向链表中下个内存段的指针。这些字段可被保存在内存段起始的两个内存位置中。例如，`length==segment[0]` 和 `segment.next==segmeng[1]`。图 12.6b（左上部分）例示了典型的 *freeList* 状态。

要分配指定大小的内存块时，该算法必须在 *freeList* 中搜索大小合适的内存段。有两种常用的启发式算法来完成此搜索任务。第一种是“**最优适应**（*Best-fit*）”**算法**，该算法在整个链表中搜索大小最匹配的内存段分配给变量。第二种是“**最先适应**（*First-fit*）”**算法**，该算法将链表中找到的第一个“大小能满足变量要求的内存段”分配给变量，一旦找到合适的段，就从中取出所需要的内存块（返回块起始位置的前一个位置，`block[-1]`，被用来保存其长度，在内存空间的去配过程中会用到该长度值）。接着，该段在 *freeList* 中被更新，成为分配之后剩余的部分。如果块中没有剩余的内存，或者若剩余部分太小，则从 *freeList* 中去掉整个段。

收回无用对象的内存块时，该算法将收回的块追加到 *freeList* 中。具体的细节如图 12.6b 所示。




```
Initialization:
  freeList = heapBase
  freeList.length = heapLength
  freeList.next = null

// 分配一个大小为 size 个字的内存块
alloc(size):
  利用最优适应法（best-fit）或最先适应法（first-fit）搜索 freeList,
  以获取一个满足 segment.length>size 条件的段。
  如果没有找到这样的段，则返回 failure
  （或者尝试整合碎片）
  block =找到的内存段中（满足条件 segment.length>size）的部分
  （如果该段剩余空间太小，block = 找到的整个内存段）
  更新 freeList 以体现内存分配情况
  block[-1] = size + 1 // 记录内存块的大小，以便之后对其进行去配
  Return block

// 释放已经完成使命的对象
deAlloc(object):
  segment = object - 1
  segment:length = object[-1]
  将 segment 插入 freeList
```

图 12.6b 改进的内存分配机制（循环利用）

经过频繁分配、收回操作之后，图 12.6b 中所描述的动态分配机制会产生内存碎片。因此，应该考虑一些“碎片整理”的操作，即将那些虽然在物理上连续而在 *freeList* 中从逻辑上被划分到不同内存段的内存区域合并起来。可以在对象被去配时，或 `alloc()` 没有找到合适内存块时执行碎片整理操作，也可依据其他条件来执行。

12.1.4 变长数组和字符串 Variable-length Arrays and Strings

假如希望支持诸如 `s1="New York"` 或 `s2=readLine ("enter a city")` 这样的高级操作，那该如何来实现这种可变长度的抽象概念呢？现代语言中的常用方法是使用 `String` 类来提供创建并操纵字符串对象的服务。在物理上，字符串对象可以通过使用数组来实现。在创建字符串时，一般为该数组分配的空间足够用来保存可能的最大长度。而字符串的实际长度可能比最大值要短，且应在字符串对象的整个生命周期中保存该实际长度值。例如，执行一条命令 `s1.eraseLastChar()`，那么 `s1` 的实际长度就由 8 变成了 7（虽然创建对象时数组的初始长度没有变）。通常，超出当前字符串长度的数组单元不会作为字符串的内容。

大多数编程语言都有字符串类型，以及其他变长数据类型。字符串对象通常由语言的标准程序库提供，比如 Java 语言中的 `String` 类和 `StringBuffer` 类，或者 C 语言中的 `strXXX` 函数。

12.1.5 输入/输出管理 Input/Output Management

计算机一般会连接到各种输入/输出设备，比如键盘、屏幕、鼠标、磁盘、网卡等。各个 I/O 设备都有其自己的机电特性和物理特性，因此在这些设备上进行数据读写操作涉及到很多技术细节。对程序员而言，高级语言利用诸如 `c=readChar()` 和 `printChar(c)` 这样的高级操作指令将这些细节隐藏（封装）起来。实际的 I/O 操作是通过操作系统提供的服务实现的。

如此一来，操作系统的重要功能之一就是要处理连接到计算机上的各种 I/O 设备。可以使用一组称为**设备驱动**（*device driver*）程序的操作系统服务来处理 I/O 底层操作，封装设备接口的物理细节，达到方便访问和使用设备的目的。本书中描述了处理两种最普遍的

I/O 设备——屏幕和键盘——的基本要件。在逻辑上将屏幕的处理分为两个独立的模块：**图形输出**处理和**字符输出**处理。

图形输出

绘制像素 现今的大多数计算机使用**光栅**（即 *raster*，也称为**位图**，*bitmap*）显示技术。在位图屏幕中所能执行的唯一基本操作就是绘制一个**像素**（*pixel*），即在屏幕上绘制用(列,行)坐标指定的"点(dot)"。一般来说，列是从左至右开始计数(即常用的 *x*-轴方向)，而行是从上往下计数（与常用的 *y*-轴方向相反）。因此屏幕的最左上角像素的坐标才是(0,0) 。

在底层进行像素绘制是属于硬件操作，它依赖于屏幕和显示卡接口的具体细节。如果屏幕接口是基于 RAM 驻留内存的**内存映像**（*memory map*）进行构建的（Hack 平台就是如此），那么将适当的二进制值写入到内存中代表该像素的内存单元，就可以完成画像素的任务（如图 12.7 所示）。

Hack 屏幕的内存映像接口在 5.2.4 小节中介绍过。根据之前的描述编写 `drawPixel` 算法是个简单的任务，作为练习留给读者来完成。既然已经知道了如何去绘制单一像素，那么接下来就让我们来了解如何绘制线条和圆。

绘制线条 要在位图屏幕上的两个位置之间画一条线，最好的方法就是用一系列的像素来逼近连接两点之间那条"理想的"直线。注意，绘制所使用的"笔"仅能沿四个方向移动，分别是上、下、左、右四个方向，因此画出来的线并不真的平直，而是曲折不平的。要让画出来的直线看起来平直一点，唯一的办法就是使用更高分辨率的屏幕。人眼的视网膜中的图像也是由"像素"网格组成，因此人眼能够分辨的图像粒度也是有限的。于是高分辨率的屏幕和打印机就能够"欺骗"人眼，让人感觉用像素或打印点组成的线条是平滑的。实际上，这些线条总是曲折不平的。

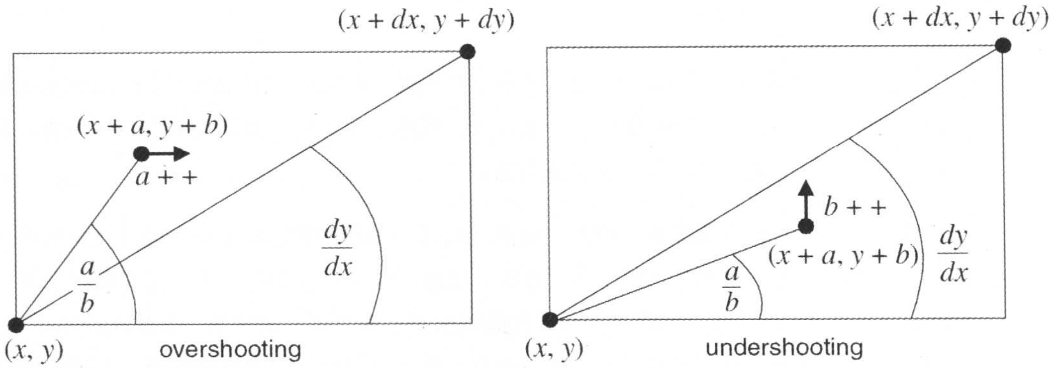
drawPixel (*x*, *y*):
// 与硬件相关
// 假设有一个屏幕的内存映像
将预定的值写入与屏幕中显示位置 location(*x*,*y*)相对应的内存单元内 *n*
location (*x*, *y*).

图 12.7 画像素点

从位置(x1, y1)到位置(x2, y2)画一条线的过程中，第一步是画(x1, y1)像素，然后以曲折地方式向(x2, y2)方向前进，一直到达终点为止。图 12.8a 展示了该过程的具体细节。

为了将线条绘制算法扩展为通用的线条绘制程序，必须考虑 *dx*,*dy*<0、*dx*>0、*dy*<0 以及 *dx*<0、*dy*>0 这些可能性。要完善线条绘制程序，还要注意该算法不能处理两种特殊情况：*dx*=0 或 *dy*=0，即画垂直和水平直线的情况。这两种常见的情况应该用单独的优化过的算法来处理。

图 12.8a 中描述的算法有个比较麻烦的问题，那就是每次循环迭代中都使用了除法操作。除法操作不仅耗时，而且还需要进行浮点运算，而非简单的整数运算。容易想到的解决方案是将 *a*/*dx*<*b*/*dy* 条件替换成等价条件 $a \cdot dy < b \cdot dx$，这样就只需要整数的乘法操作。另外，仔细观察等价条件 $a \cdot dy < b \cdot dx$ 的代数结构，会发现其实根本不必要使用任何乘法操作就能进行大小比较的判断。如图 12.8b 所示，可以通过维护一个额外的变量来达到目的，每当 *a* 或 *b* 的值改变时，就将更新的 $a \cdot dy - b \cdot dx$ 差值存放至该变量。




```
drawLine(x, y, x + dx, y + dy):
  // 假设 dx, dy > 0
  initialize (a, b) = (0, 0)
  while a<=dx and b<=dy do
    drawPixel(x + a, y + b)
    if a/dx < b/dy then a++ else b++
```

图 12.8a 画线

```
// 维护一个额外变量 adyMinusbdx 用于测试条件 a/dx < b/dy 是否成立，
// 并测试变量 adyMinusbdx 是否变成负数。
初始化:        set adyMinusbdx = 0
执行 a++时:    set adyMinusbdx = adyMinusbdx + dy
执行 b++时:    set adyMinusbdx = adyMinusbdx − dx
```

图 12.8b　仅使用某些附加操作就可以提高检测效率

圆的绘制　在位图屏幕上画圆有很多方法。这里介绍一种算法（如图 12.9 所示），该算法利用之前已经实现的 3 个程序（**乘法**、**平方根**和**线条绘制**）来实现。

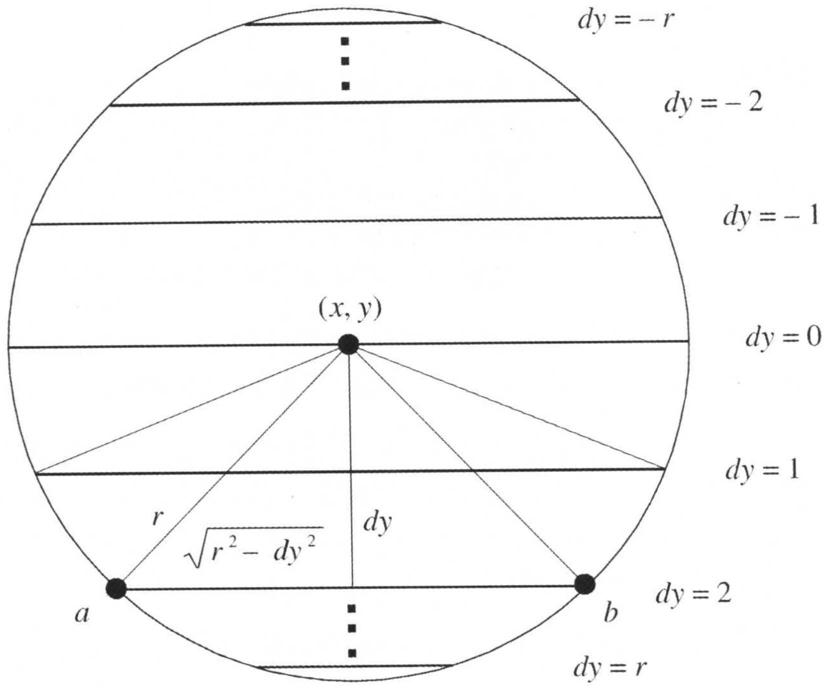
该算法的基本思想是，在屏幕上画一系列水平线（如图 12.9 中的 *ab* 直线），每行直线的范围是 $y-r$ 到 $y+r$。因为 r 是用像素来描述的，所以该算法会沿着圆的纵轴（从上至下的方向）在屏幕的每一行画一条直线，最后形成完全填充的圆。对该算法作小小的改动就可以实现绘制空心圆的算法。

要注意的是，该算法效率不太高，因为每次迭代中的平方根运算是个耗时的操作。有很多更高效地画圆算法，包括那些仅使用加法操作的算法（与这里介绍的画线算法的指导思想是一致的）。

字符输出　到目前为止，描述的所有的输出都是图像输出：像素、直线、圆。现在来介绍**字符**是如何通过使用操作系统提供的良好服务，用一个一个的像素在屏幕上打印出来。下面介绍实现细节。

为了在位图屏幕上显示文本，首先必须将物理上基于像素点的屏幕，在逻辑上以字符为单位划分成为若干区域，每个区域能输出单个完整的字符。对于 256 行 512 列的屏幕，如果为每个字符分配一个 11*8 像素的网格，那么屏幕上就总共能显示 23 行，每行 64 个字符（还有 3 行像素没有使用）。

接下来，可以为需要显示在屏幕上的字符设计漂亮的**字体**（*font*），然后利用一系列字符位图来绘制字体。例如，图 12.10 画出了字母'A'的一种位图字体。



point $a = (x - \sqrt{r^2 - dy^2}, y + dy)$ point $b = (x + \sqrt{r^2 - dy^2}, y + dy)$

drawCircle(x, y, r):
  for each $dy \in -r \ldots r$ do
    drawLine from $(x - \sqrt{r^2 - dy^2}, y + dy)$ to $(x + \sqrt{r^2 - dy^2}, y + dy)$

图 12.9 画圆

需要注意的是，为了使该显示机制能考虑必要的字符间距，必须保证每个 11*8 的字符位图中包含了与下个字符之间至少要有 1 个像素的间隔并且与邻近行之间至少要有 1 个像素的间隔（具体间隔大小可能会随着具体各字符大小的不同而不同）。

字符通常是从左至右，被一个接一个地显示在屏幕上。比如，两个命令 `print("a")` 和 `print("b")` 可能意味着程序员希望看到 `"ab"` 出现在屏幕上。因此写字符的控制程序必须维持一个“光标（cursor）”对象，让其指示下一个字符在屏幕上应该出现位置。

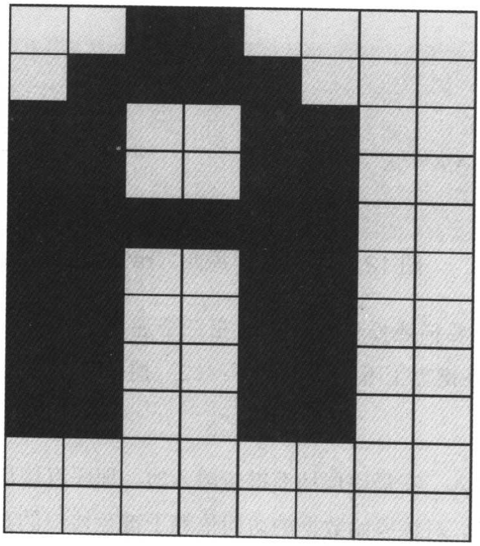


图 12.10 字母“A”的位图（bitmap）

光标信息由行计数和列计数组成。对于前面描述的那个 23 行*64 字符构成的字符屏幕，可以用 $0 \le line \le 22$ 和 $0 \le column \le 63$ 来表示。要在位置(*line*, *column*)上画一个字符，就将字符的位图写到行 $line \cdot 11 \ldots line \cdot 11+10$ 和列 $column \cdot 8 \ldots column \cdot 8+7$ 构成的像素区域。画完字符之后，光标应该向右移动一格（即 *column=column*+1）；需要换新行时，*row* 应该加 1 并将 *column* 置为 0。对于如何处理到达屏幕底部时的情况，通常的解决办法是执行“滚动（scrolling）”操作。另一个可能的办法是在屏幕左上角重新开始，即将光标重置为(0, 0)。

到目前为止，知道了如何在屏幕写字符。其他类型的数据的绘制则基于这个基本方式来实现：**字符串**按照一个字符一个字符来绘制，而**数字**则先被转换成字符串，然后以字符串的形式来绘制，等等。

键盘处理 在处理支持用户输入的文本操作时，所需要考虑的问题更多，而不仅只是屏幕上看到的内容。比如对于命令 `name=readLine("enter your name:")`，其底层实现可不简单，因为它涉及到对不可预见事件的处理：在代码结束之前，用户会按下键盘上的一些键（具体按下了哪些键，这是无法事先预见的）。问题在于，用户按键持续的时间各不一样。因此，最好的办法就是将所有这些繁琐的底层处理都封装在诸如 `readLine` 这样的 OS 程序当中，让高级程序免于纠缠这些细节。

```
keyPressed( ):
//与具体的键盘接口相关
if 键盘上有键被按下
      return 该键对应的 ASCII 值
else
      return 0
```

图 12.11 捕获“原始（raw）”键盘输入

这一部分主要描述操作系统如何在三个递增的抽象层级上来管理面向文本的输入：（1）侦测当前在键盘上哪个键被按下；（2）捕获单字符输入；（3）捕获多字符输入，也就是字符串。

侦测键盘输入 在最底层捕获键盘输入时，程序直接从硬件获取数据，并确定用户当前按下了哪个键。对该原始数据的访问依赖于键盘接口的具体特性。比如，若接口是被键盘持续更新的内存映像（Hack 平台正是如此），就可以简单地通过检查内存映像中对应的内存单元的内容来确定哪个键被按下。该算法实现如图 12.11 所示。

举例来说，如果你知道主机中键盘映像在内存中的具体地址，那么该算法的实现就相当于内存查找操作而已。

读取单一字符 在“键被按下”事件和“键被释放”事件之间的时间是不可预测的。因此必须编写相应的代码来处理这种时间上的差异。另外，当用户在键盘上按键的时候，通常要给用户提供可视化反馈，以便提示用户哪个键被按下了（你可能认为这是理所当然的）。通常在屏幕中下一个将要输入字符的位置显示光标，当字符被输入后（也就是对应的键被按下之后），在光标指示位置就会显示输入字符的位图作为反馈。图 12.12 中实现了这种逻辑。

读取字符串 通常，用户输入多个键的内容只有在按下了回车键（产生**换行符**）之后才会被认为输入结束。而且，在按回车键之前，应该允许用户按退格键（backspace）来清

```
readChar():
  // 读取并返回一个单字符，同时
  显示光标
  while 没有按下任何键时
    什么都不做 // 等待，直到有键被按下
  c = 当前按下的键所对应的代码
  while 某个键被按下时
    什么都不做 // 等待用户释放按键
  print c （在当前光标位置列印）
  将光标右移一格
  return c
```

```
readLine():
  // 读取并回显“一行”
  // （到遇到 newline 换行符为止）
  s = 空字符串
  repeat
    c = readChar()
    if c = 换行符 newline
      print 换行符
      return s
    else if c = 退格符
      从 s 中移除最后一个字符
      将光标左移一格
    else
      s = s.append(c)
```

图 12.12、图 12.13 捕获“受控的（cooked）”键盘输入

除之前输入的字符。图 12.13 中的代码实现了这种逻辑和可视化效果。

跟前面一样，处理输入的方法是基于一系列级联的分层抽象。高级程序依赖于 `readLine` 抽象，`readLine` 依赖于 `readChar` 抽象，而 `readChar` 又依赖于 `KeyPressed` 抽象，`KeyPressed` 则依赖于硬件。

12.2 Jack OS 规范详述 The Jack OS Specification

前一节介绍了一系列经典算法，并用它们实现了一组典型的操作系统任务。本节将正式地以 API 形式来描述 Jack 操作系统（Jack OS）。Jack OS 也可被看作是 Jack 程序语言的扩展，因此描述文档原样复制了第 9.2.7 节中的“Jack 标准程序库”。第 9 章的 OS 规范详述主要是面向使用抽象服务的程序员；而本章的 OS 规范详述则主要面向实现抽象服务的程序员。具体的技术信息和实现技巧在第 12.3 节进行详细阐述。

Jack 操作系统由如下 8 个类组成：

- *Math* 提供基本的数学运算；
- *String* 实现字符串 String 类型和字符串相关操作；
- *Array* 实现数组 Array 类型和数组相关操作；
- *Output* 处理屏幕上的文本输出；
- *Screen* 处理屏幕上的图像输出；
- *Keyboard* 处理键盘的用户输入；
- *Memory* 处理内存操作；
- *Sys* 提供与程序执行相关的服务。

12.2.1 Math

该类实现各种数学运算操作。

- function void **`init`** ()：仅供内部使用；
- function int **`abs`** (int x)：返回 *x* 的绝对值；
- function int **`multiply`** (int x, int y)：返回 *x* 与 *y* 的乘积；
- function int **`divide`** (int x, int y)：返回 *x/y* 除法结果的整数部分；
- function int **`min`** (int x, int y)：返回 *x* 和 *y* 中的较小值；
- function int **`max`** (int x, int y)：返回 *x* 和 *y* 中的较大值；
- function int **`sqrt`** (int x)：返回 *x* 的平方根的整数部分。

12.2.2 String

该类实现字符串 `String` 数据类型以及与字符串相关的操作。

- constructor String **`new`** (int maxLength)：构建新的空字符串（长度为 0），最多能包含 maxLength 个字符；
- method void **`dispose`** ()：清除字符串；
- method int **`length`** ()：返回字符串的长度；
- method char **`charAt`** (int j)：返回字符串第 j 个位置上的字符；
- method void **`setCharAt`** (int j, char c)：将字符串中第 j 个元素置为字符 c；
- method String **`appendChar`** (char c)：在字符串末尾追加字符 c 并返回整个字符串；
- method void **`eraseLastChar`** ()：删除字符串中最后一个字符；

- method int **intValue** ( )：返回字符串（或是“从最左边开始直到遇到非数字字符为止”的子串）的整数值；
- method void **setInt** (int j)：以字符串形式保存 *j* 所代表的整数；
- function char **backspace** ( )：返回退格字符（backspace）；
- function char **doubleQuote** ( )：返回双引号（"）字符；
- function char **newLine** ( )：返回换行符。

12.2.3 Array

该类构造和清除数组。

- function Array **new** (int size)：构造大小为 size 的新数组；
- method void **dispose** ( )：清除数组。

12.2.4 Output

该类提供在屏幕上打印文本的服务。

- function void **init** ( )：仅供内部使用；
- function void **moveCursor** (int i, int j)：将光标移动到第 *i* 行的第 *j* 列，并且删除此位置上的字符；
- function void **printChar** (char c)：在光标处打印字符 *c* 并在该行将光标向前移动一格；
- function void **printString** (String s)：在光标处开始打印字符串 *s*，并将光标向前移到打印结束的位置；
- function void **printInt** (int i)：在光标处开始打印整数 *i*，并且将光标向前移动到打印结束的位置；
- function void **println** ( )：将光标移动到下一行的起始处；
- function void **backspace** ( )：在该行将光标向后移动一格。

12.2.5 Screen

该类提供在屏幕上绘制图形的服务。列索引从 0 开始，从左至右递增。行索引从 0 开始，自上而下递增。屏幕的尺寸大小依赖于硬件（Hack 平台上的屏幕尺寸为 256 行×512 列）。

- function void **init** ( )：仅供内部使用；
- function void **clearScreen** ( )：清屏；
- function void **setColor** (boolean b)：为后续 `drawXXX` 命令设置绘图颜色（white = false，

black=true）；

- function void **drawPixel** (int x, int y)：在坐标(x, y)处绘制像素；
- function void **drawLine** (int x1, int y1, int x2, int y2)：在像素点(x1, y1)与像素点(x2, y2)之间画一条直线；
- function void **drawRectangle** (int x1, int y1, int x2, int y2)：绘制有填充色的长方形，长方形的左上角坐标是(x1, y1)，右下角坐标是(x2, y2)；
- function void **drawCircle** (int x, int y, int r)：绘制圆心坐标为(x, y)，半径为 r（r<=181）的有填充色的圆。

12.2.6 Keyboard

该类提供从标准键盘上读取输入的服务。

- function void **init** ( )：仅供内部使用；
- function char **keyPressed** ( )：返回当前键盘上被按下的键所对应的字符；如果当前没有键被按下则返回 0；
- function char **readChar** ( )：等待键盘上某个键被按下后又被释放，将该键的字符返回并显示到屏幕上；
- function String **readLine** (String message)：从键盘输入读取一行字符串（直到遇见换行字符为止），在屏幕上显示该行字符串，最后返回该字符串。该函数能处理退格（back-space）；
- function int **readInt** (String message)：从键盘输入读取一行字符串（直到遇见换行字符为止），在屏幕上显示该行字符串，然后返回其对应的整数值（直到遇见第一个非数字字符为止）。该函数也能处理退格。

12.2.7 Memory

该类提供直接访问宿主平台的主内存的服务。

- function void **init** ( )：仅供内部使用；
- function int **peek** (int address)：返回地址为 address 的内存单元中的内容；
- function void **poke** (int address, int value)：将整数值 value 存入地址为 address 的内存单元中设置在这个地址上的主内存的值；
- function Array **alloc** (int size)：从内存堆（heap）中寻找并分配一块大小为 size 的内存区，返回其基地址的指针；

- function void **deAlloc** (Array o)：收回之前分配给对象的内存空间。

12.2.8 Sys

该类提供与程序执行相关的服务。

- function void **init** ( )：调用其他 OS 类的 `init` 函数，进而调用 `Main.main()` 函数。仅供内部使用；
- function void **halt** ( )：中止程序执行；
- function void **error** (int errorCode)：在屏幕上打印错误代码，并中止程序执行；
- function void **wait** (int duration)：等待大约 duration 毫秒后返回。

12.3 实现 Implementation

上一节描述的操作系统能以一组 Jack 类的集合来实现。每个 OS 子程序都能以 Jack 构造函数、函数或方法的形式来实现。第 12.2 节已经给出了所有子程序的 API，第 12.1 节已经介绍了相关的算法。本节将提供一些提示和建议以便协助你完成 OS 实现。12.5 节给出了最后的一点技术细节以及用于对所有 OS 服务进行单元测试的测试程序。需要注意的是，在操作系统 API 中给出的大部分子程序都是相当简单的，很容易通过 Jack 编程来实现，因此这里只讨论其中的一部分 OS 子程序的实现。

OS 中的有些类可能需要初始化。比如对于某些数学函数，如果能够利用上之前计算的结果值，运行速度就会更快。这些值可以被保存在静态数组里，该数组只用被初始化一次就可被 `Math` 类中所有程序共用。通常，在需要为 OS 类 `Xxx` 编写一段初始化代码时，应该将这段初始化代码封装到名为 `Xxx.init()`的函数中。本节后续部分会解释在计算机启动和引导 OS 时如何唤起这些 `init()`函数。

12.3.1 Math

Math.multiply(),Math.divide(): 图 12.1 和图 12.2 中的算法只能用来操作非负整数。处理负数的方法之一是在运算过程中用绝对值进行计算，然后再适当地设置符号。乘法不需要这么做：如果乘数与被乘数是以 2 进制补码形式表示，那么乘积就是正确的，不用再作任何额外处理。

注意图 12.1 所示的算法，在每次迭代当中，第二个数的第 j 位都被提取出来。建议将该操作封装到函数 **bit(x,j)** 中：如果整数 x 的第 j 位是 1，则返回 true；否则返回 false。

其实利用比特位移动操作就很容易实现 `bit(x,j)` 函数。可惜 Jack 目前并不支持比特位移动操作，因此作罢。为了提高该函数在 Jack 实现中的运行效率，可以定义一个定长为 16 的静态数组，比如 `twoToThe[j]`，其第 j 个位置保存着以 2 为幂、以 j 为指数的值。该静态数组只需被初始化一次（在 `Math.init` 中进行初始化即可），然后通过比特位的布尔操作在 `bit(x, j)` 的实现中利用此数组。

在图 12.2 中，y 以 2 为因数作乘法，直到 $y>x$ 为止。需要考虑的细节是，y 可能溢出。可以通过检查 y 是否为负来侦测溢出的情况。

Math.sqrt(): 由于图 12.3 中 $(y+2^j)^2$ 的计算可能发生溢出，因此计算结果可能是负数（这属于异常结果）。可通过改变算法的 *if* 逻辑来有效地解决此问题：

$$\text{if } ((y+2^j)^2 \le x) \text{ and } ((y+2^j)^2 > 0) \quad \text{then} \quad y = y+2^j$$

12.3.2 String

如 12.1.4 小节所述，字符串对象能以数组的形式来实现。同样地，所有字符串相关的服务都能用字符串操作来实现。有个重要实现细节是，在整个操作中必须维护字符串的实际长度，超过实际长度的数据项不应该被当作字符串的有效部分。

String.intValue(), String.SetInt(): 这两个函数可以分别使用图 12.4 和图 12.5 中的算法来实现。注意，这两个算法都未讨论处理负数的情况，但在实现中你必须处理出现负数的情况。

`String` 类中的所有其他程序都很简单。注意，换行符、退格符以及双引号的 ASCII 码分别为 128、129 和 34。

12.3.3 Array

要注意 `Array.new()` 并不是构造函数，而是个普通的函数（尽管其名字很像构造函数）。因此，应该通过对 `Memory.alloc()` 函数的调用来显式地为新数组分配内存空间。同样地，应该调用 `Memory.deAlloc()` 函数来显式地释放数组占用的内存空间。

12.3.4 Output

字符位图 建议采用之前讨论过的 11*8 像素的字符位图，从而形成 23 行 64 列的字符屏幕。为所有可打印的 ASCII 字符设计并构建位图是一件工作量很大的事情，为此本书配套网站提供了预定义的位图（其中特意留下一两个字符未实现，留给读者作为练习）和由 Jack 代码编写的 `Output` 类的骨干程序实现，该类中定义了为每个可打印的 ASCII 码保存其位图的数组（以此创建字体）。该数组包含 11 个数据项，每个数据项对应一行像素。数据项 j 的值是二进制数，该二进制数的 8 个比特位代表了位图中第 j 行所显示的 8 个像素。

12.3.5 Screen

Screen.drawPixel(): 在屏幕上绘制像素是通过调用 `Memory.peek()` 和 `Memory.poke()` 函数直接操作屏幕的内存映像中特定内存单元来实现的。前面说过，在 Hack 平台上的屏幕内存映像规定了在 c 行和 r 列上 $(0 \le c \le 511, 0 \le r \le 255)$ 的像素被映射到内存位置 $16384+r*32+c/16$ 的 c%16 位。绘制一个像素点需要在访问的字中改变一个比特位，这可以在 Jack 语言中通过比特位操作来实现。

Screen.drawLine(): 图 12.8a 中的算法可能会导致溢出。图 12.8b 中的改进算法解决了溢出问题。

Screen.drawCircle(): 图 12.9 中的算法也可能会导致溢出。将圆的半径限制到 181 以内就能避免溢出问题。

12.3.6 Keyboard

在 Hack 平台中，键盘的内存映像是位于内存地址 24576 上的 16-比特位的字。

Keyboard.keyPressed()：该函数提供了对映像内存单元的直接访问，可利用 Memory.peek()来实现。

Keyboard.readChar()，**Keyboard.readString()**：这些函数提供了对单一字符输入和字符串输入的“加工后（即有选择暴露某些部分作为对外接口，而不是完全暴露整个内部结构）的”访问。图 12.12 和图 12.13 分别给出了算法实现。

12.3.7 Memory

Memory.peek()，**Memory.poke()**：这两个函数提供对底层内存的直接访问。这如何用高级语言完成呢？实际上，Jack 语言包含一个后门，使程序员能获得对计算机内存的完全控制权。通过简单的 Jack 编程，可以利用这种黑客技巧来实现 peek 和 poke 函数。

该技巧的实质其实是以非常规的方式来使用引用变量（即指针）。Jack 语言并没有禁止程序员将常数赋给指针变量，因此该常数可被当作是内存的绝对地址。当该地址恰好是某个数组的基地址时，这种手法就能奏效，使得对整个计算机的内存进行方便和直接的访问成为可能（即将整个物理内存空间看作一个一维数组）。图 12.14 展示了具体细节。

图 12.14 中前面两行代码将 memory 数组的基地址指向计算机 RAM 中的第一个地址。为了设置或读取物理地址为 j 的 RAM 单元的值，只要操纵数组的数据项 memory[j]就可以了。这会使得编译器去操纵地址为 0+j 的内存单元，这正是我们所期望的效果。

```
// 为 RAM 创建 Jack 层级的“代理”
var Array memory;
let memory = 0;
// 如此一来我们就可以使用如下的代码
let x = memory[j] // 其中 j 是任意 RAM 地址
let memory[j] = y // 其中 j 是任意 RAM 地址
```

图 12.14 在 Jack 语言中应用后门来控制整个内存空间

在前面指出过，Jack 语言在编译期并不从堆中为数组分配空间，而是在运行期间，当数组的 new 函数被调用时才进行内存空间分配。然而，new 函数的初始化无法达到目的，因为整个技巧的关键是要将数组定位在一个特定的地址上，而不是让 OS 在堆中为其分配一个我们无法控制的地址。简言之，我们不经过“正常”手段来为该数组分配内存空间，这种技巧达到了我们的目地。

Memory.alloc()，**Memory.deAlloc()**：这两个函数可以通过图 12.6a 的基本算法来实现，也可以通过图 12.6b 的改进算法来实现，使用“最优适应”法或“最先适应”法来实现。前面介绍过在 Hack 平台上 VM 的实现标准指定了堆所占用的内存空间为 2048～16383。

12.3.8 Sys

Sys.init()：用 Jack 语言编写的应用程序就是一组类的集合。其中一个类必须被命名为 Main，该类必须包含一个名为 main 的主函数。为了使应用程序开始运行，Main.main() 函数应该被首先调用。要知道操作系统本身就是一个程序（即一组类），所以当计算机启动时，首先开始运行的是操作系统程序，而且是操作系统的主程序。

于是，这个命令序列应该按如下顺序来执行：首先，VM（第 8 章）包含了启动代码，该代码能自动调用 Sys.init() 函数，该函数存在于 OS 的 Sys 类中，再由该函数调用操作系统的其他所有类的 init() 函数，最后调用存在于应用程序中的主函数 Main.main()。

Sys.wait()：该函数应该根据 Hack 平台的实际条件限制，务实地实现出来。特别值得指出的是，你可以在函数返回之前使用一个运行时间大约为 *n* 毫秒的循环。必须根据你的计算机设置好时间进度，以便实现以 1 毫秒为单位的等待，因为这个常数是因 CPU 而异的。如此一来，你的 Sys.wait() 函数将是不可移植的，但这就是现实生活——不可能事事完美。

Sys.halt()：该函数通过无限循环来实现。

12.4 观点 Perspective

本章提供的软件程序库包含了多数操作系统里的一部分基本服务，如内存管理、I/O 驱动、初始化处理、提供没有在硬件中实现的数学函数，以及实现诸如 *string* 抽象之类的数据类型。本书将这个标准程序库称为“操作系统”，以此来反映其主要功能，即将繁琐的硬件细节和硬件特性封装到透明的软件包中，使得高层程序员能够通过简单的接口来使用它提供的服务。当然，Jack 的操作系统与工业级强度的操作系统仍然有很大差距。

作为十分初级的操作系统，Jack OS 还缺少一些与操作系统最紧密相关的基本功能。比如，我们的 OS 不支持多线程，也不支持并行处理；相比之下，大多数操作系统的关键任务就是要支持这些功能。我们的 OS 也没有大容量存储设备；相比之下，其他操作系统处理的是文件系统抽象。我们的 OS 既没有“命令行”接口（比如在 Unix Shell 和 DOS 窗口中的命令行），也没有图形化接口（比如窗口、鼠标、图标等）；相比之下，这些都是用户期望操作系统应该具备的功能，以便于用户与其进行交互。很多其他的常见服务也没有出现在 Jack OS 中，比如安全机制、通信机制等。

另一个与多数操作系统的主要区别在于 OS 代码和用户代码之间的交互。在大多数计算机中，OS 代码被认为是“享有特权的”，硬件平台禁止用户代码执行各种操作，只允许 OS 代码在其上的运行。因此，要获取操作系统的服务就需要具有比简单的函数调用要复杂得多的机制。此外，编程语言通常将这些 OS 服务包含在一般的函数或方法中。相比之下，在 Hack 平台中，OS 代码和用户代码是没有区别的，操作系统服务运行在相同的“用户模式”下，就像运行应用程序一样。

在效率方面，介绍的乘法和除法的算法是标准实现。这两个算法及其变体都一般都是在硬件中实现。这些算法的运行时间是 $O(n)$次加法操作。由于将两个 n-比特位的数相加需要 $O(n)$-比特位的操作（硬件中的门电路），因此这些算法最终需要进行 $O(n^2)$ -比特位的操作。 还有一些乘法和除法算法的运行时间比 $O(n^2)$ 快得多，而且对于较大的位数，这些

算法的效率会更高。同样的，对于本书介绍的一些几何操作（比如绘制线条和画圆），其优化版本也经常是在特殊的图形加速硬件中实现的。

感兴趣的读者可以自己动手，扩展这些 OS 功能，欢迎大家这么做（这正是我们在第 13 章中的建议）。

12.5 项目 Project

目标 实现本章所描述的操作系统。可以按照任意顺序，单独地实现各个 OS 类并对其进行单元测试。

资源 本项目中的主要工具是用于开发 OS 的 Jack 编程语言。因此，你还需要用配套提供的 Jack 编译器以及测试程序来编译和测试你的 OS 实现。为了能够对 OS 进行单元测试，你需要 OS 的完整编译版本，包含一组 `.vm` 文件（每个文件对应一个 OS 类）。最后，你还需要用到配套提供的 VM 仿真器，以此作为进行实际测试的平台。

约定 编写 Jack OS 实现，然后使用本项目描述的程序和测试场景（testing scenarios）来进行测试。每个测试程序都涉及到对 OS 服务的某个子集的测试。

测试方法

建议单独地实现各个 OS 类，并对其进行单元测试。可以首先编译你编写的 OS 类，然后将得到的 `.vm` 文件置于同一个路径下，该路径要包含配套提供的 OS 其他部分的 `.vm` 文件。为了能够单独地开发、编译并测试每个 OS 类 `Xxx.jack`，建议按照下面的步骤来进行：

1. 将下列文件放到同一路径下：1）你正在开发的 OS 类 `Xxx.jack`；2）所有配套提供的 OS `.vm` 文件；3）相关的测试程序（若干个 `.jack` 文件）。

2. 利用提供的 Jack 编译器来对该路径进行编译。编译器会编译你的 `Xxx.jack` OS 类和测试程序的类文件。在处理过程中会创建新的 `xxx.vm` 文件，并取代原来提供的 OS 类。这正是我们所期望的结果，于是该路径现在应该包含可执行的测试程序、完整的 OS（除了原来旧的 `Xxx.vm` OS 类之外），以及你的 `Xxx.vm` 。

3. 将该路径下的代码（OS 和测试程序）加载到 VM 仿真器中。

4. 执行代码，然后根据下面给出的描述，检查 OS 服务是否工作正常。

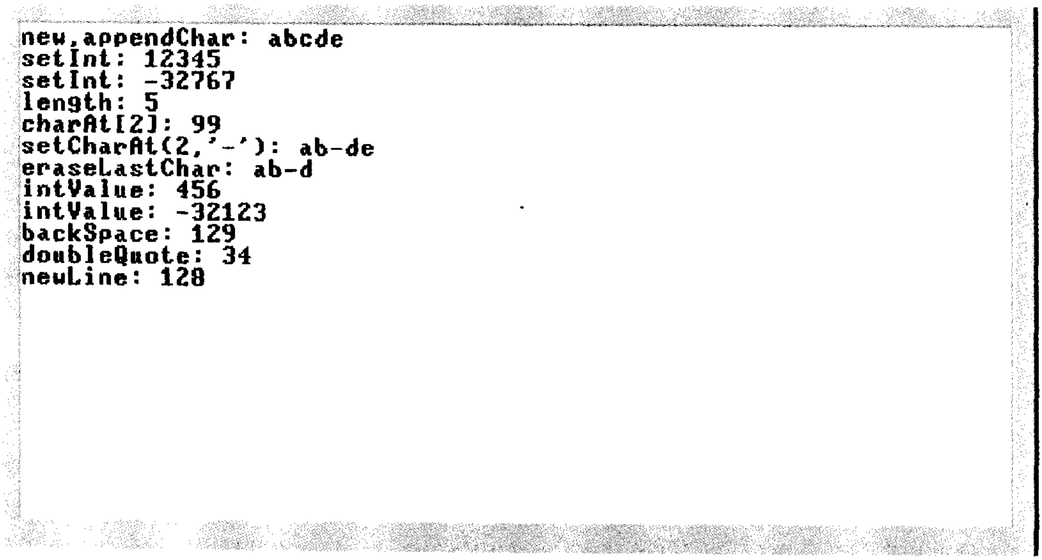
OS 类和测试程序

OS 中有 8 个类：`Memory`、`Array`、`Math`、`String`、`Output`、`Screen`、`Keyboard` 和 `Sys`。对于每个 OS 类 `Xxx` 本书都提供了骨干程序文件 `Xxx.jack`，其中包含了所有必要的子程序声明；另外还提供了相应的名为 `Main.jack` 的测试类，以及相关的测试脚本。

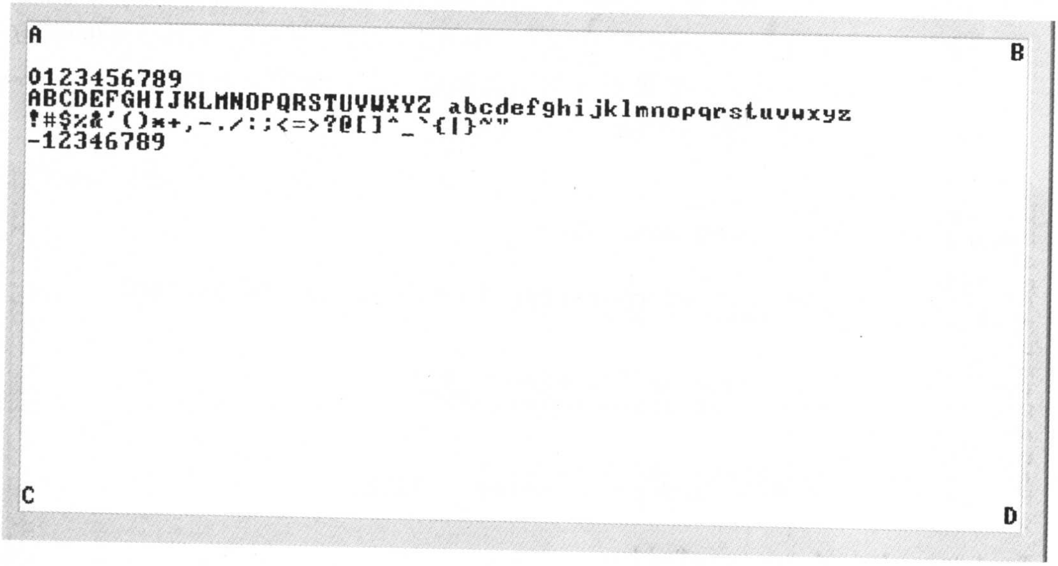
Memory，Array，Math 要测试你的 OS 类实现，请对相关路径进行编译，并在 VM 仿真器上执行测试脚本，并确认结果文件与比较文件之间的比对成功。

需要注意的是，本书提供的测试程序没有包含针对 `Memory.alloc()` 和 `Memory.deAlloc()` 函数的完整测试。对这两个内存管理函数的完整测试需要通过观察内部的实现细节来完成，而这些细节在用户级测试中是不可见的。因此建议在 VM 仿真器中利用单步调试来测试这两个函数。

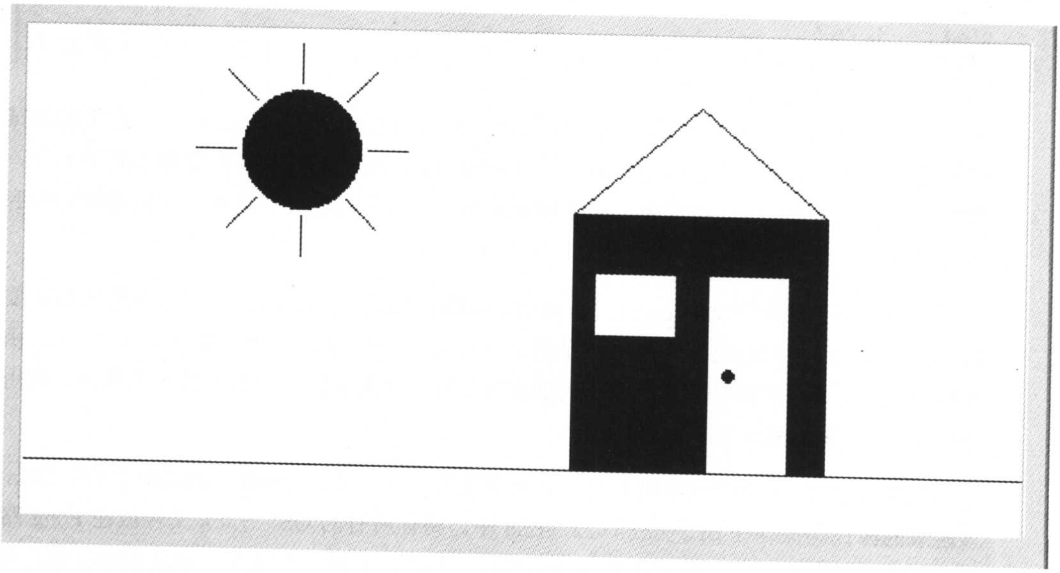
String 执行测试程序后应该产生如下输出：



Output 执行测试程序后应该产生如下输出：

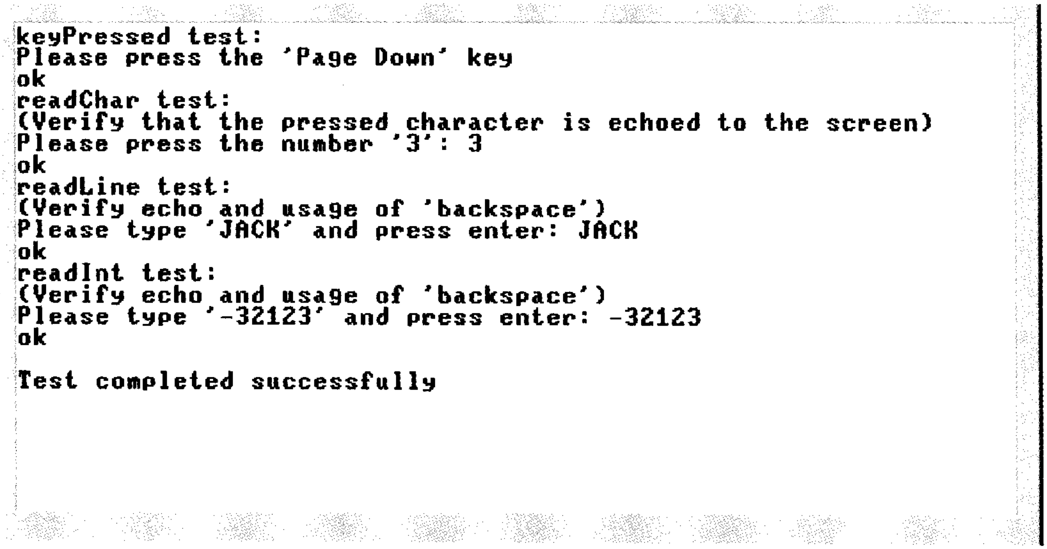


Screen 执行测试程序后应该产生如下输出：



Keyboard 该 OS 类需要用户与程序进行交互的测试程序来达到测试目的。

对于 Keyboard 类中的每个函数(keyPressed(),readChar(),readLine(),readInt()),该程序要求用户在键盘上按下一些键。如果函数被正确地执行了，所需的键也按下了，那么程序会打印文本“ok”，然后继续测试下一个函数。如果不是这样，程序会对相同的函数要求进行重复测试。如果所有的测试都成功了，程序会打印“Test ended successfully”，如下面的屏幕截图所示。



Sys 该类中只有两个函数可以进行测试：Sys.init()和 Sys.wait()。提供的测试程序要求用户按下任意键，然后等待两秒（利用 Sys.wait()），接着在屏幕上打印另外一条信息来测试 Sys.wait()函数。从键被释放到下一条信息被打印出来之间的时间间隔应该是 2 秒钟。

本书并不单独地测试 Sys.init()函数。前面介绍过，Sys.init()函数执行所有必要的 OS 初始化并接着调用每个测试程序的 Main.main()函数。因此，除非 Sys.init()函数能够执行正确，否则任何程序都将无法正确工作。单独测试该函数的简单方法是，利用你的 Sys.vm 文件来运行 *Pong* 游戏。

完整的测试 对每个 OS 类进行单元测试之后，利用 *Pong* 游戏来测试整个 OS 实现，该游戏的源代码保存在 projects/12/Pong 中。将你所有的 OS .jack 文件放在 Pong 路径下，编译该路径，然后在 VM 仿真器中执行。如果游戏能正常运行，那就该恭喜你！对于这个完全由你自己编写的操作系统，你是其当之无愧的主人！

第 13 章 后记：发掘更多乐趣

Postscript: More Fun to Go

We shall not cease from exploration, and at the end we will arrive where we started, and know the place for the first time.

永远不应停止探索的脚步，直到最后，再次抵达起点，我们才刚刚开始了解脚下的路。

——T. S. Eliot（1888～1965），诗人

祝贺你！你已经完成了完整的计算机系统的构建工作。希望你喜欢这次旅程。请让本书的作者与你分享一个小秘密：我们创作这本书更是一种享受。毕竟，要**设计**这个计算机系统，而设计通常都是项目中“最有意思”的部分。本书作者希望你们中的一部分人——那些有冒险精神的读者——能够加入到设计者的行列中来。也许你希望改进这个体系结构；也许你对在某些部分添加新功能有些好想法；也许你在构想一个更广泛强大的系统。接下来，也许你希望作为领航员，自己决定往哪个方向前进，而不再只是搞清楚如何到达目的地。

对你在各个项目中所编写的软件进行修改和扩展，就可以实现很多可供选择的设计要素。比如，对于汇编语言、Jack 语言和操作系统，你可以通过改变其规范，重写汇编编译器、高级语言编译器和 OS 实现，来进行修改和扩展。其他一些改变就可能需要对我们提供的软件进行修改了。比如，如果你想要改变 VM 规范或者硬件规范，就要修改仿真器。或者如果你想要在 Hack 计算机中添加新的输入输出设备，你就可能需要在硬件仿真器中将它们作为内置芯片进行建模。

为了让读者能够进行自由地修改和扩展，我们公开了本书相关软件的所有源代码。除了用于在 Windows 和 Linux 平台上启动软件的批处理文件之外，所有的代码都是用 Java 编写的。关于软件及其文档，都可以在本书的网站（*http://www.idc.ac.il/tecs*）上获得。

只要你认为有必要，欢迎对我们提供的所有工具进行修改和扩展。如果你愿意的话，还可以将你的心得与其他人分享。希望我们所编写的代码以及相关文档能够为修改和扩展过程提供良好的基础和便利。特别要提到的是，本书提供的硬件仿真器具有简单且清晰的接口，有利于添加新的“内置（built-in）”芯片。你可以通过该接口使用诸如磁盘存储或通信设备这样的芯片来扩展虚拟硬件平台。

无法想象你的设计改进会是什么样的。接下来只简要地阐述一下我们所考虑的一些改进方案。

13.1 硬件的实现 Hardware Realizations

本书所介绍的所有硬件模块的实现都是基于软件并用 HDL 模拟出来的。事实上，这正是实际当中设计硬件的方法。然而，有时这些 HDL 设计会涉及到硅片级的处理，付诸工艺实施，从而成为“真实的计算机”。如果让 Hack 或者 Jack 也运行在某个由“真实的材料”制造而成的 “真实的平台”上，岂不是更好吗？有很多不同的方法来实现这个目标。极端的方法是，利用现存的 Hack 的 HDL 设计，处理好与 RAM、ROM 和 I/O 设备相关的具体实现问题，来亲身实践真实芯片的制造过程。另一个极端的方法就是，在一些现存的硬件设备（比如手机或 PDA）上进行 Hack、VM，或是 Jack 平台的）仿真。当然，在这样的项目中，可能都会需要缩小 Hack 的屏幕尺寸，以便保证合理的分配和使用有限的硬件资源。

13.2 硬件的改进 Hardware Improvements

虽然 Hack 是**存储程序计算机**（*stored program computer*），但其所运行的程序必须被预存在 ROM 设备中。在当前的 Hack 体系中，没有办法在用户控制下将另一个程序加载到计算机中，只有模拟整个物理 ROM 芯片的替代芯片除外。若希望在不作彻底改变的前提下为计算机加入“加载程序”的能力，就可能需要修改整个体系中的若干层级。应该修

改 Hack 硬件，以便让加载的程序保存在可写的 RAM 中，而不是 ROM 中。应该将某种类型的永久存储（比如片上磁盘，disk-on-chip）添加到硬件结构中，以便支持程序存储。还应该扩展操作系统，以便操控永久存储设备以及用于加载和运行程序的新的逻辑设计。这样的话， 就有必要提供 OS 用户界面了（“shell”或“DOS 窗口”之类的设施）。

13.3 高级语言
High-Level Languages

跟其他领域的专业人员一样，程序员对他们所使用的工具（编程语言）有着深厚感情，喜欢通过某种方式把它变得更为个性化。Jack 语言的确还有很多不太令人满意的地方，可以对其进行改进或者甚至是用新的语言来替代它。有些改进比较简单，有些则很棘手，还有些改变可能需要修改 VM 规范（比如让语言支持继承）。

13.4 优化
Optimizations

本书几乎完全回避了优化问题（只在第 12 章中考虑了一些效率问题）。优化是黑客大展身手的地方。你可以首先在现有编译器或者硬件中进行局部优化（在现有的平台中， 对 VM 翻译器进行优化就是很可行的方案）。在更大的范围内进行优化则涉及到改变接口规范（比如机器语言或 VM 语言等）。

13.5 通信
Communications

如果 Hack 计算机能够连接到 Internet 上该多好！要做到这点，可以将通信芯片追加到硬件结构中，然后编写 OS 代码来处理该芯片以及处理更高层级的通信协议。还需要编写一些其他的程序来与仿真的通信芯片进行“交谈”，以此提供与 Internet 的接口。比如说，用 Jack 来编写基于 HTTP 的 Web 浏览器就是可行且有实践价值的方案之一。

以上就是本书作者对于设计的一些想法——你有什么好的想法呢？

附录 A： 硬件描述语言（HDL）

Hardware Description Language

Intelligence is the faculty of making artificial objects, especially tools to make tools.

智能即制造人造物——特别是制造“制造工具的工具”——的能力。

——Henry Bergson（1859 ~ 1941），哲学家

硬件描述语言（*Hardware Description Language*，HDL）是一种用于定义和测试芯片的规范：芯片对象的接口是由负责传输布尔信号的输入以及输出管脚构成，而对象本体则是由互联的底层芯片组成。本附录描述了一种典型的 HDL，它能被与本书配套的硬件仿真器所理解。第 1 章（特别是第 1.1 小节）中提供了一些重要的背景知识，本附录正是建立在这些背景知识的基础上。

如何使用本附录 本附录实际上是技术参考文档，因此读者没有必要从头读到尾。我们建议根据您自己的需要有选择地进行阅读。HDL 是一门表述直白且容易理解的语言，因此，最好的学习方法就是利用硬件仿真器来编写一些 HDL 程序。建议你从下面的例子开始，尝试着编写 HDL 程序。

A.1 范例 Example

图 A.1 描述了一种芯片，它能够对两个 3-比特位的数值进行比较，并输出比较结果（即相等或不相等）。该芯片逻辑利用 Xor 门对这 3 个比特位进行比较，如果所有位都相同则输出为真。每个被 HDL 程序调用的内部函数 Xxx 都代表一个独立的芯片，被定义在单独的 `Xxx.hdl` 程序中。所以，对于编写 `EQ3.hdl` 程序的芯片设计者而言，要始终假设程序中使用的 3 个底层程序（`Xor.hdl`，`Or.hdl` 和 `Not.hdl`）都是可以现成使用的。更重要的

是，芯片设计者无需考虑这些芯片的底层实现细节。当用 HDL 构建新的芯片时，那些参与设计的内部芯片总是被看作为黑盒子，于是设计者只用关注如何在构建芯片的过程中去正确地组织这些内部芯片。

```
/**检查两个3-bit位输入总线上的数值是否相等*/
CHIP EQ3 {
   IN a[3], b[3];
   OUT out; // True iff a=b
   PARTS:
   Xor(a=a[0], b=b[0], out=c0);
   Xor(a=a[1], b=b[1], out=c1);
   Xor(a=a[2], b=b[2], out=c2);
   Or(a=c0, b=c1, out=c01);
   Or(a=c01, b=c2, out=neq);
   Not(in=neq, out=out);
}
```

图 A.1 HDL 程序范例

幸亏有了 HDL 这种模块特性，使得对于所有 HDL 程序，甚至那些功能复杂的高级芯片，都能以一种简洁易读的形式保存下来。例如，一个复杂的 RAM16K 芯片可以由若干内部芯片（比如 RAM4K 芯片）来组成，而每个内部芯片仅用一条 HDL 语句就可以描述。在硬件仿真器模拟芯片功能的过程中，它会按照芯片的体系结构，自顶向下递归地将芯片扩展为成千上万个互联的基本逻辑门。然而，芯片设计者不必关心这些细节，他只需考虑芯片高层的抽象体系结构就行了。

A.2 约定

Conventions

文件扩展名：每个芯片都定义在一个独立的文本文件中。即名为 Xxx 的芯片被定义在文件 `Xxx.hdl` 中。

芯片结构：芯片的定义由**描述头**（*header*）和**描述体**（*body*）组成。描述头定义了芯片的**接口**（*interface*），它充当芯片 API 或公共文档的角色。描述体定义其**实现**（*implementation*）。当设计者将该芯片作为内部芯片来构建更高级的芯片时，就不用关心其实现。

语法约定：HDL 是区分大小写的。HDL 关键字用大写字母表示。

标识符命名：芯片和管脚的名称是可以由任意字母和数字组成的序列，但不能以数字开头。芯片名称通常以大写字母开头，管脚名称通常以小写字母开头。基于可读性的考虑，这些名称可以包含大写字母。

空格：空格符、换行符和注释都将被忽略。

注释：HDL 支持下列的注释格式：

```
// 注释直至行末
/* 注释包围的内容 */
/** API 文档注释 */
```

A.3 将芯片加载到硬件仿真器 Loading Chips into Hardware Simulator

HDL 程序（即芯片的形式描述）可通过三种方式被加载到硬件仿真器中。首先，用户可以通过交互式方式进行，应用“load file”菜单或 GUI 图标来打开 HDL 文件。其次，包含 `load Xxx.hdl` 命令的测试脚本也可以达到相同的效果。其三，每当 HDL 程序被加载并解析时，根据芯片递归的体系结构，仿真器都会根据程序中所使用的 Xxx 芯片去加载其对应的 `Xxx.hdl` 文件。无论采用何种方式，仿真器都将经历以下逻辑流程：

```
if Xxx.hdl 存在于当前的路径下
  then 将该芯片（以及由其派生的所有芯片）加载到仿真器中
else
  if Xxx.hdl 存在于仿真器的 builtIn 芯片路径下
    then 将该芯片（以及由其派生的所有芯片）加载到仿真器中
  else
    提示出错信息
```

仿真器的 `builtIn` 路径下包含了本书所介绍的所有芯片（除最高级的 CPU、Memory 和 Computer 芯片之外）的可执行版本。因此，即使你没有亲手实现底层的芯片，也仍然可以构建和测试任意一种芯片：仿真器会自动为你调用 `bulitIn` 目录下所需要的底层芯片的内置实现版本。同样地，如果你已经自己动手用 HDL 实现了某一底层芯片 Xxx，那么当你将其对应的 `Xxx.dhl` 文件从当前目录下移除时，仿真器同样会自动为你从 `bulitIn` 目录下调用该芯片的内置实现版本。最后，在某些情况下用户（而不是仿真器）可能希望直接加载一个内置芯片，只需要进入 `tools/builtIn` 路径（该目录是硬件仿真器环境的一个部分）然后选择你需要的芯片就行了。

A.4 芯片描述头（接口）Chip Header (Interface)

HDL 程序的描述头具有如下格式：

```
CHIP 芯片名 {
   IN 输入管脚名, 输入管脚名, ...;
   OUT 输出管脚名, 输出管脚名, ...;
   // 这里是描述体。
}
```

- **CHIP 声明**（*CHIP declaration*）：芯片名称紧接着关键字 `CHIP` 之后。它的所有 HDL 代码都置于花括号之中。
- **输入管脚**（*Input Pins*）：关键字 `IN` 之后是一组用逗号分隔的输入管脚名称。最后以分号结束。
- **输出管脚**（*Output Pins*）：关键字 `OUT` 之后是一组用逗号分隔的输出管脚名称。最后以分号结束。

输入和输出管脚所传输的数据宽度默认为 1-比特位。多位总线（bus）以 *pin name*[*w*] 的形式来声明（比如，`EQ3.hdl` 中的 `a[3]`）。这种定义方式表示该管脚是数据宽度为 *w*-比特位的总线。总线中的每个位从右至左以索引号 0...*w*−1 来表示（即索引号 0 代表最低位 LSB）。

A.5 芯片描述体（实现）Chip Body (Implementation)

A.5.1 单元 Parts

典型的芯片由若干底层芯片组成，这些底层芯片的输入/输出管脚按照特定的“逻辑”（连接部分 connectivity pattern）相互连接，从而实现高级芯片的功能。而这些“逻辑”就是由 HDL 程序员按以下格式在芯片描述体内编写的：

```
PARTS:
内部芯片单元语句;
内部芯片单元语句;
...
内部芯片单元语句;
```

其中每个**内部芯片单元语句**都描述了一个内部芯片及其所有同其他芯片的连接信息，语法如下：

```
chip name(connection,…,connection);
芯片名 (连接, …, 连接);
```

其中每个连接（connection）使用如下语法描述：

```
part's pin name = chip's pin name;
单元的管脚名 = 芯片的管脚名
```

在本附录中，当前定义的芯片称为芯片（*chip*），PARTS 部分列出的底层芯片称为单元（*part*）。

A.5.2 管脚和连接
Pins and Connections

根据芯片的定义，每个**连接**（connection）描述了一个单元的管脚如何同其他单元的管脚相连接的。最简单的情况是，程序员将一个单元的管脚与芯片的一个输入或输出管脚相连接。大多情况下，可能一个单元的管脚和另一个单元的管脚相连接。为了描述这种内部连接信息，需要引入**内部管脚**（*internal pin*）的概念。

内部管脚 为了将一个单元的输出管脚与其他单元的输入管脚连接起来，HDL 程序员可以创建和使用内部管脚（比如用 *v* 表示），它按以下方式描述：

```
Part1 (..., out = v);        //单元 1 的输出连接到 v
Part2 (in = v, ...);         //单元 2 的输入和 v 连接
Part3 (a = v, b = v, ...);   //单元 3 的 a 和 b 两个输入管脚和 v 连接
```

按照需要，一旦在 HDL 程序中创建了某一指定的内部管脚（比如 *v*）后，那么今后在对它的使用过程中就无需特别声明。每个内部管脚都有一个输入和多个输出（数量不限），也就是说：它仅能从单一信号源接收信号，却可以将其输出信号作为多个单元的输入（通过多个连接信息）。在前面的例子中，内部管脚 *v* 就将输出信号同时作为 Part2（通过 in）和 Part3（通过 a 和 b）的输入。

输入管脚 一个单元的输入管脚可以由以下信号源作为输入:

- 芯片的一个输入管脚
- 一个内部管脚
- 常数 `true` 或 `false`（分别代表 1 和 0）

每个输入管脚都只有一个输入，意味着它仅能与一个信号源相连。因此 `Part (in1=v,in2=v,...)` 是一条正确的语句，而 `Part (in1=v,in1=u,...)` 则不正确。

输出管脚 单元的每个输出管脚可以以下信号端作为输出:

- 芯片的一个输出管脚
- 一个内部管脚

A.5.3 总线
Buses

每一个在连接中所使用的管脚，无论是输入、输出或内部管脚，都可能是**多位总线**（*multi-bit bus*）。输入和输出管脚的宽度（bit 位）定义在芯片的描述头中。而内部管脚的宽度可以通过它们的连接推导出来。

为了连接一个输入或输出管脚中某些特定的数据位，可以使用语法 `x[i]` 或 `x[i...j]=v`（其中 x 是管脚名，v 是内部管脚）来实现。该语句意味着管脚 *x* 中只有位于索引号 *i* 到 *j* 之间的数据位才连接到指定的内部管脚 *v* 上。内部管脚（比如上面的 *v*）不该有索引号，其数据宽度就是与它相连的总线管脚的数据宽度，并且在今后的 HDL 程序中，它的数据宽度将保持该值不变。

常数 `true` 和 `false` 也可以被用作总线，其总线宽度可以通过连接信息的上下文得到。

范例

```
CHIP Foo {
   IN in[8]       //8-比特位输入
   OUT out[8]    //8-比特位输出
   //Foo 的描述体
}
```

现在假设其他芯片通过以下语句来调用芯片 `Foo`：

```
Foo (in[2..4]=v, in[6..7]=true, out[0..3]=x, out[2..6]=y)
```

其中 v 是先前定义的 3-比特位的内部管脚。在以上语句的连接信息中：`in[2..4]=v` 和 `in[6..7]=true` 将 `Foo` 芯片的 `in` 总线的某些数据位设置为下列值：

in:	7	6	5	4	3	2	1	0	(Bit)
	1	1	?	v[2]	v[1]	v[0]	?	?	(Contents)

现在假设 `Foo` 芯片的输出为：

out:	7	6	5	4	3	2	1	0
	1	1	0	1	0	0	1	1

在这种情况下，连接信息中的 `out[0..3]=x` 和 `out[2..6]=y` 会产生以下值：

x:	3	2	1	0
	0	0	1	1

y:	4	3	2	1	0
	1	0	1	0	0

A.6 内置芯片
Built-In Chips

硬件仿真器提供内置芯片库，库中的芯片可作为其他芯片的内部单元来使用。内置芯片可通过如 Java 编程语言编写代码来实现，其实现细节将被标准 HDL 接口所屏蔽。所以，内置芯片有标准的 HDL 描述头（接口），但是其 HDL 描述体（即实现）是内置的。图 A.2 给出了典型的例子。

关键字 `BUILTIN` 之后的标识符是实现芯片逻辑功能的程序单元（program unit）名称。当前版本的硬件仿真器是由 Java 编写的，所有的内置芯片都是以经过编译的 Java 类来实现的。因此，内置芯片的 HDL 描述体格式如下：

`BUILTIN` *Java class name;*

其中 *Java class name* 是实现该芯片功能的 Java 类的名称。通常，该类与其所代表的芯片具有相同的名称，例如，`Mux.class` 。所有的内置芯片（即编译后的 Java 类文件）都保存在名为 `tools/builtIn` 的目录下，该目录也是仿真器环境的标准组成部分。

内置芯片提供三种特定的服务：

- **基础**（*fundation*）：某些芯片是构建所有其他芯片的基本原子（atoms）。例如，我们用基本 Nand 门和 flip-flop 门来分别构建组合芯片和时序芯片。因此，硬件仿真器中包含了 `Nand.hdl` 和 `DFF.hdl` 的内置版本。

```
/** 16-bit Multiplexor.
If sel = 0 then out = a else out = b.
该芯片有一个通过外部 Java 类实现的内置实现版本 */
CHIP Mux16 {
   IN a[16], a[16], sel;
   OUT out[16];
   BUILTIN Mux; // 指向 builtIn/Mux.class 的引用,
                // 该文件实现了 Mux.hdl 和 Mux16.hdl 这两个内置芯片
}
```

图 A.2 内置芯片的 HDL 定义

- **验证和效率**(*certification and efficiency*):要对复杂芯片进行模块化的开发,一种方法是首先完成其底层芯片的内置实现。这使得芯片设计者可以专注于构建和测试芯片的逻辑功能,而无需考虑底层芯片的实现细节,因为仿真器会自动调用已经实现的底层芯片。此外,使用芯片的内置实现机制来代替在程序中通过 HDL 实现芯片的方法,也是很有意义的,因为在模拟过程中,前者在速度和内存空间方面的优势要比后者明显。比如,将 `RAM4K.hdl` 加载到仿真器中时,仿真器会在内存中创建一个由成千上万个底层芯片(按照芯片结构自顶向下递归到 flip-flop 门)组成的数据结构。很明显,我们每次将 `RAM4K` 作为高级芯片的内部单元调用时,没有必要去重复这种模拟工作。最实用的小技巧是:为了提升性能并且尽量减少错误,要尽可能使用芯片的内置芯片实现。

- **可视化**(*visualization*):对于某些高级芯片,若能直接查看其操作(比如内存单元)状况,就更利于理解和调试这些芯片。为此,内置芯片都额外配备了 GUI 显示功能。每当这些芯片被加载到仿真器中,或作为被加载芯片的底层单元被调用时,其 GUI 都会在屏幕上显示出来。除了可视化效果之外,具有 GUI 显示功能的芯片与其他芯片的特性以及使用方法都是相同的。具有 GUI 显示功能的芯片细节在 A.8 小节阐述。

A.7 时序芯片 Sequential Chips

计算机芯片分为两种：**组合**（*conbinational*）**芯片和时序**（*sequential*）**芯片**（也称为**时钟芯片**，*clocked chip*）。组合芯片的操作具有即时性（instantaneous）。当用户或测试脚本改变组合芯片的一个或多个输入管脚的值时，这些输入值的变化立刻就会被仿真器反映（仿真器通过芯片的内部逻辑对新输入值进行计算）到芯片的输出管脚上。相比之下，时序芯片的操作受时钟的控制。当改变时序芯片的输入值时，受时钟的控制，输入的改变只有在下一个时钟周期才反映到芯片的输出管脚上。

事实上，即使时序芯片（比如计数器芯片）的输入不发生任何变化，但随着时间的变化，其输出值也可能改变。相比之下，组合芯片不会因为时间的流逝而改变它们的输出值。

A.7.1 时钟 The Clock

仿真器通过两个名为 *tick* 和 *tock* 的操作来对时间的流逝进行建模。这两个操作可以模拟一系列时钟周期或**时间单元**（*time unit*），每个时钟周期都由两个阶段（phase）组成：*tick* 代表时钟周期中第一阶段的结束和第二阶段的开始，*tock* 则代表下一个时钟周期中第一阶段的开始。在这个过程中实际流逝的时间与模拟无关，因为我们能够完全控制时钟。换言之，仿真器的用户或测试脚本都可以任意地制定 *ticks* 和 *tocks* 操作，从而让时钟生成任意长度的时钟周期序列。

两部分的时钟周期按照如下方式来规范在模拟芯片中所有时序芯片的操作。在时钟周期的第一阶段（*tick*），模拟芯片中每个时序芯片读取各自的输入值，并且根据模拟芯片的组合逻辑来生成该芯片一个新的内部状态。在时钟周期的第二阶段（*tock*），模拟芯片的输出被赋予新值。因此，如果我们“从外部”看一个时序芯片，会发现它的输出管脚仅在 *tocks*（连续的时钟周期之间）才稳定到一个新值。

有两种方式来控制模拟时钟：手动控制方式和基于脚本的控制方式。首先，仿真器的 GUI 中有一个时钟形状的按钮。在该按钮上点一下就结束时钟周期的第一个阶段（一个 *tick*），接着点一下则结束时钟周期的第二个阶段（一个 *tock*），并接着开始下一个时钟周

期的第一个部分。另外，我们还可以通过测试脚本来模拟时钟，比如利用命令 `repeat n {tick, tock, output;}`。这个例子指示仿真器运行 *n* 个时钟周期，并且在此过程中打印某些值。测试脚本以及命令 `repeat` 和 `output` 将在附录 B 中进行详细介绍。

A.7.2 时钟芯片和管脚 Colcked Chips and Pins

内置芯片可以利用下面的语句显式地定义其时钟依赖性：

```
CLOCKED 管脚, 管脚, ..., 管脚;
```

其中每个**管脚**都是在芯片描述头中声明的输入管脚或输出管脚。如果在 CLOCKED 列表中包含一个**输入管脚** *x*（*input pin x*），这就是告诉仿真器，直到下一个时钟周期开始为止，输入管脚 *x* 的改变不应该影响芯片的任何输出管脚的值；如果在 CLOCKED 列表中包含一个**输出管脚** *x*（*output pin x*），这就是告诉仿真器，直到下一个时钟周期开始为止，芯片任何输入管脚的变化都不应该影响芯片输出管脚 *x* 的值。

这里要注意，实际上很可能一个芯片只有特定的一些输入或输出管脚与时钟相关。如果是这样，那些与时钟无关的输入管脚的变化就会以组合逻辑的方式来即时地影响那些与时钟无关的输出管脚。实际上，也有可能 CLOCKED 关键字之后的管脚序列为空，这意味着即使芯片内部状态的变化依赖于时钟的变化，但对任意输入管脚的改变都会立即反映到它的输出管脚上。

芯片的“时钟（clocked）”属性 仿真器如何知道芯片是与时间相关的呢？如果芯片是内置的，那么其 HDL 代码会包含 CLOCKED 关键字。如果芯片不是内置的，那么当它的一个或多个底层单元是时钟芯片时，我们也认为该芯片是与时间相关的。一般方法是按照芯片的结构，自顶向下的递归检测底层芯片的“时钟”属性，直到发现其中一个内置芯片显式地具有时间相关性。如果找到这样一个芯片，那么它使得所有构建于其上的芯片都继承了对时间的相关性。因此芯片的 HDL 代码中并没有任何信息可以来指示它是否具有时钟属性——唯一的确认方法就是阅读芯片的文档。下面就让我们来看看内置 DFF 芯片（图 A.3）是如何来影响其他芯片的“时钟”属性的。

在我们的计算机体系结构中，每个时序芯片都多少以这样或那样的方式依赖于 DFF 芯片。例如，RAM64 芯片是由 8 个 RAM8 芯片构成。每个 RAM8 芯片由 8 个底层 Register

```
/** D-Flip-Flop (D 触发器) .
If load[t-1]=1 then out[t]=in[t-1] else out does not change. */
CHIP DFF {
  IN in;
  OUT out;
  BUILTIN DFF; // 由 builtIn/DFF.class 实现
  CLOCKED in, out; // 以显式的方式调配时钟
}
```

图 A.3 时序芯片的 HDL 定义

芯片构成。每个 Register 芯片由 16 个 Bit 芯片构成。而每个 Bit 芯片中包含了一个 DFF 单元。于是 Bit、Register、RAM8、RAM64 以及所有构建在它们之上的内存组件都将是时钟芯片。

不要忘记，时序芯片也可能包含不受时钟影响的组合逻辑电路。例如，每个时序 RAM 芯片中，都包含用来管理取址的逻辑组合电路（见第 3 章）。

A.7.3 反馈环
Feedback Loops

当芯片中某个内部单元的输出直接或间接地影响了该单元的输入时，那么对该芯片的使用将形成反馈环（feedback loop）。比如，考虑下面两个直接反馈的例子：

```
Not ( in=loop1, out=loop1 )     // 错误
DFF ( in=loop2, out=loop2 )     // 有效
```

在这两个例子中，一个内部管脚（`loop1` 或 `loop2`）试图将芯片的输出作为芯片的输入，从而构成一个环路。两个例子的区别在于 `Not` 是一个组合逻辑芯片而 DFF 是一个时序性（clocked）芯片。在上例的 Not 中，`loop1` 在 `in` 和 `out` 之间形成了即时的、无法控制的相关性，也称**数字竞争**（*data race*）；而在 DFF 中，由 `loop2` 形成的 `in-out` 相关性被 DFF 的时钟逻辑延迟，因此 `out(t)` 不是 `in(t)` 的函数，而是 `in(t-1)` 的函数。

有效/错误的反馈环 当仿真器加载芯片时，它会递归地检查该芯片内部的各种连接是否构成了反馈环路。对于每个环路，仿真器会检查该环路是否在某处与时钟管脚相连。

如果相连，那么这个环路将允许存在。否则，仿真器会停止处理并发出错误提示信息。这么做是为了避免发生不受控制的数字竞争。

A.8 芯片操作的可视化 Visualizing Chip Operations

某些内置芯片可以支持“GUI 功能”。这些芯片具有可视的效果，从而使芯片操作变得形象具体。支持 GUI 功能的芯片可以通过两种不同方式来参与模拟过程，就像任何其他芯片一样。首先，用户可以将其直接加载到仿真器中。其次，当将一个具有 GUI 功能的芯片当作被模拟芯片的一个内部单元时，仿真器会自动调用它。在两种情况下，仿真器都会在屏幕上显示芯片的图像。此图像就是交互式的 GUI 组件，可以利用它来观察芯片当前的内容以及它内部状态的改变。当前版本的仿真器中，支持 GUI 功能的芯片如下：

ALU：显示 Hack 的 ALU 的输入和输出，以及当前被计算的函数。

Register（有三种类型：地址寄存器 ARegister，数据寄存器 DRegister，以及程序计数器 PC)：显示寄存器的内容，并且允许修改内容。

内存芯片（ROM32K 和各种 RAM 芯片)：显示类似于数组的滚动图像，显示了所有内存位置的内容，并且允许对这些内容进行修改。如果内存位置的内容在模拟过程中发生了变化，那么在 GUI 中相应的数据项也会改变。在 ROM32K 芯片（它作为我们的计算机平台的指令内存）中，它的 GUI 还有一个按钮，它允许从外部文本文件中加载机器语言程序。

Screen **芯片**：如果被加载芯片的 HDL 代码调用了内置的 Screen 芯片，硬件仿真器会显示 256 行 512 列的窗口来模拟物理屏幕。当 RAM 中的屏幕内存映射在模拟过程中发生了变化，那么在屏幕 GUI 中相应的像素也会通过仿真器实现中嵌入的“刷新”控制来改变。

Keyboard **芯片**：如果被加载芯片的 HDL 代码调用了内置的 Screen 芯片，仿真器会显示一个可按的键盘图标。点击该按钮会将你的计算机的实际键盘与模拟的芯片联系起来。

```
// 提供 GUI 显示功能支持的芯片范例。
// 该芯片的逻辑是无实际意义，仅用来让仿真器显示其他芯片的 GUI 效果
CHIP GUIDemo {
  IN in[16], load, address[15];
  OUT out[16];
  PARTS:
  RAM16K(in=in, load=load, address=address[0. .13], out=a);
  Screen(in=in, load=load, address=address[0. .12], out=b);
  Keyboard(out=c);
}
```

图 A.4 具备 GUI 支持的芯片的 HDL 定义

于是，在实际键盘上每次按下的键都会被模拟的芯片获取，它的二进制值会显示在键盘的内存映射中。如果用户将光标移到仿真器 GUI 的其他区域，那么对键盘的控制又交回实际的计算机。图 A.4 中显示了很多这里介绍的特性。

图 A.4 中的芯片逻辑将 16-位 `in` 值传到两个目的地：RAM16K 芯片的 *address* 管脚和 Screen 芯片的 *address* 管脚（根据这些芯片的文档，编写这段代码的 HDL 程序员可以计算出 address 管脚的宽度)。此外，该芯片逻辑会将当前按下的键盘的键值发送到内部管脚 `c` 。执行这些看上去没有太大意义的操作只有一个目的，即说明仿真器如何来处理内置的具有 GUI 功能的芯片的。图 A.5 给出了实际效果。

A.9 已经提供的内置芯片与新的内置芯片
Supplied and New Built-In Chips

图 A.6 中列出了硬件仿真器所提供的内置芯片。这些基于 Java 语言的芯片实现支持 Hack 计算机平台的构建和模拟（虽然其中的一些芯片也可以支持其他的 16-位平台）。若

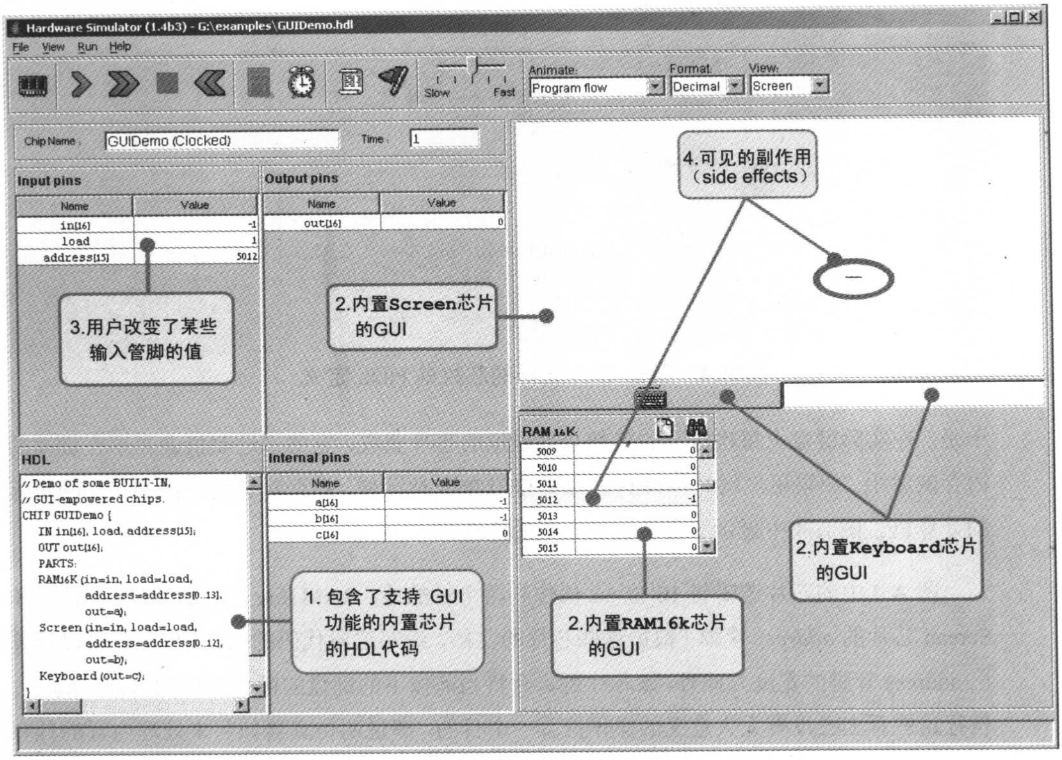


图 A.5 支持 GUI 功能的芯片，因为加载的 HDL 程序内部使用了支持 GUI 功能的芯片（图中的第 1 步），所以仿真器就画出了对应的 GUI 图像（图中的第 2 步）。当用户改变芯片输入管脚值的时候（图中的第 3 步），仿真器也会在对应的 GUI 中反映出这些变化（图中的第 4 步）。图中圆圈内的水平线是将−1 存储到地址为 5012 内存单元中的可视化效果。因为−1 的 16-位 2-补码为 1111111111111111，所以计算机从第 156 行的第 320 列开始画出 16 个像素，这恰好是对应于屏幕内存映像中内存单元（地址为 5012）的屏幕坐标（第 4 章详述了内存与屏幕之间的映射）

芯片名	描述章节	是否有GUI	说　明
Nand	1		所有组合芯片的基本组件
Not	1		
And	1		
Or	1		
Xor	1		
Mux	1		
DMux	1		
Not16	1		
And16	1		
Or16	1		
Mux16	1		
Or8way	1		
Mux4way16	1		
Mux8way16	1		
DMux4way	1		
DMux8way	1		
HalfAdder	2		
FullAdder	2		
Add16	2		
ALU	2	√	
Inc16	2		
DFF	3		所有时序芯片的基本组件
Bit	3		
Register	3		
ARegister	3	√	与 Register 操作相同，支持 GUI
DRegister	3	√	与 Register 操作相同，支持 GUI
RAM8	3	√	
RAM64	3	√	
RAM512	3	√	
RAM4K	3	√	
RAM16K	3	√	
PC	3	√	程序计数器
ROM32K	5	√	GUI 允许加载一个文本中的程序
Screen	5	√	GUI 与一个模拟的屏幕相关联
Keyboard	5	√	GUI 与实际的键盘相关联

图 A.6　当前版本的硬件仿真器提供的所有内置芯片，它们都有 HDL 接口，但以可执行 Java 类的方式被实现

你希望开发与 Hack 不同的其他硬件平台，硬件仿真器对新型内置芯片的兼容性会为你提供便利。

开发新的内置芯片 硬件仿真器能够执行任何用 HDL 编写的芯片逻辑；利用芯片扩展的 API，它也能执行用 Java 语言编写新的内置芯片（除了那些列在图 A.6 中的芯片之外）。用户可以利用 Java 来为内置芯片来添加新的硬件组件，引入 GUI 功能，加快执行速度，对正在设计中的芯片行为进行模拟，当然这些芯片还没有用 HDL 真正实现（在设计新的硬件平台以及相关的硬件构建项目时，能够模拟硬件行为的功能是很必要的）。想要了解更多关于新内置芯片开发的信息，可以参阅第 13 章。

附录 B： 测试脚本语言

Test Scripting Language

Mistakes are the portals of discovery.

错误是发现探索的温床

——James Joyce（1882 ~ 1941），小说家

测试是系统开发过程中非常重要的环节，然而在当今计算机科学的教育中却很少被关注。本书会非常认真地对待测试问题。实际上，我们坚信在着手开发新的硬件或软件模块 *P* 之前，首先应该开发用于测试它的模块 *T*。而且，模块 *T* 应该成为模块 *P* 官方开发协议（offical development's contract）的组成部分。

在实际工程中，对一个全新开发模块的最终测试方案，应该由拟定系统接口的系统设计人员制定，而不是模块的开发人员。因此，对于书中的每个芯片和软件，都提供了由我们编写的官方测试程序。虽然我们允许你采用任何适当的方式来测试自己的模块，但是为了遵守开发协议，你的模块最终还是要通过我们提供的测试。

全书各项目当中涉及到大量的测试，为了将全书所有测试的定义与执行步骤联系起来，我们设计了统一测试脚本语言。该语言在本书提供的所有仿真器中的工作方式几乎都是一样的：

- **硬件仿真器**：用于模拟和测试用 HDL 实现的芯片
- CPU **仿真器**：用于模拟和测试用机器语言编写的程序
- VM **仿真器**：用于模拟和测试用 VM 语言编写的程序

每个仿真器都带有丰富的 GUI，利用图形化的图标、批处理或测试脚本，用户能够以交互的方式来测试被加载到仿真器上的芯片或程序。**测试脚本**（*test script*）就是一个命令序列，这些命令：（a）将一个硬件或软件模块加载到相关的仿真器上；（b）让模块经历一系列预先计划好的（而不是自组织的）测试用例。此外，测试脚本还有用于打印测试结果的命令，以及将结果与期望结果（定义在提供的比较文件中）进行比较的命令。

总的来说，测试脚本能够为底层代码提供一个系统的、可复制的、用文档记录的测试，这种能力对于任何硬件或软件开发项目而言都是相当重要的。

注意 我们并不要求学生们自己编写测试脚本。**本书中所有硬件和软件模块所需的测试脚本都由我们提供，并可以在本书的网站上下载**。因此，本附录的主要目的是解释测试脚本的语法和逻辑。

B.1 文件的格式和用法 File Format and Usage

本书中任何一个用来测试硬件或软件模块的仿真器都会涉及到以下四种类型的文件：

`Xxx.yyy`：其中 `Xxx` 是模块名称，`yyy` 可能是 `hdl`、`hack`、`asm` 或 `vm`，分别代表用 HDL 编写的芯片、用 Hack 机器语言编写的程序、用 Hack 汇编语言编写的程序或者用 VM 虚拟机器语言编写的程序；

`Xxx.tst`：该测试脚本定义了一系列测试步骤，仿真器会根据这些步骤来对 `Xxx.yyy` 中的代码进行测试；

`Xxx.out`：可选的**输出文件**（*output file*），用来保存模拟结果。

`Xxx.cmp`：可选的**比较文件**（*compare file*），用来保存预先拟定的模拟结果。该结果可以说是开发人员对硬件或软件功能的期望。

所有这些文件应该被保存在相同的目录下，方便起见，我们将该目录命名为 `xxx`。在所有仿真器中，“current directory（当前目录）”是指在仿真器环境中，最新近打开文件的所在目录。

空格：空格符、换行符以及测试脚本（`Xxx.tst` 文件）中的注释都会被忽略。测试脚本不区分大小写（除文件和路径名之外）。

注释：测试脚本所支持的注释格式如下：

```
// 注释至行末
/* 注释包围的内容 */
/** API 文档注释 */
```

用法：在本书所有项目中，`Xxx.tst`、`Xxx.out` 和 `Xxx.cmp` 文件都由配套的软件包提供。这些文件用来测试 `Xxx.yyy`（它们可以算是工程开发的精髓）。有时我们还提供 `Xxx.yyy` 的骨干程序，比如省略实现部分的 HDL 接口等。项目中所有文件都是普通文本文件，并可以使用普通的文本编辑器来浏览和编辑。

通常，在开始模拟工作的时候，首先应将 `Xxx.tst` 脚本文件加载到相关的仿真器中。脚本文件中最开始的几条命令通常会指示仿真器去加载 `Xxx.yyy` 文件中的代码，然后根据需要，初始化必要的输出文件和比较文件。脚本中的其他的命令则规定了实际的测试步骤，我们在下面会详细的介绍。

B.2 在硬件仿真器中测试芯片 Testing Chips on the Hardware Simulator

本书提供的硬件仿真器是用来测试和模拟用硬件描述语言（HDL）编写的芯片。第 1 章提供了一些关于芯片开发和测试的基本背景知识，因此建议你应该首先阅读第 1 章的相关内容。

B.2.1 范例 Example

图 B.1 中的脚本用于测试在图 A.1 中定义的 EQ3 芯片。测试脚本的开头通常是一些初始化命令，接着是一连串**模拟步骤**（*simulation steps*），每个步骤都以分号结尾，它通常指示仿真器将某些指定的测试值赋予芯片的输入管脚，然后应用这些测试值对芯片逻辑进行测试之后，将指定的结果写入指定的输出文件。图 B.2 给出了 `EQ3.tst` 脚本的实际范例。

B.2.2 数据类型和变量 Data Types and Varialbles

数据类型 测试脚本支持两种数据类型：整型和字符串。整型常数可以采用十六进制（前缀为`%X`）、二进制（前缀为`%B`），或者默认的十进制（前缀为`%D`）来表示。这些值将被翻译成对应的 2-补码。例如，`set a1 %B1111111111111111`，`set a2 %XFFFF`，`set a3 %D-1`，

```
/* EQ3.tst: 测试EQ3.hdl程序。
  若EQ3芯片两个3-bit位的输入值相等则返回true，否则返回false. */
load EQ3.hdl,             // 将HDL程序加载到仿真器
output-file EQ3.out,      // 测试结果输出到该文件
compare-to EQ3.cmp,       // 比较结果输出到该文件
output-list a b out;      // 以下每条输出命令应该打印变量a, b和out的值
set a %B000, set b %B000, eval, output;
set a %B111, set b %B111, eval, output;
set a %B111, set b %B000, eval, output;
set a %B000, set b %B111, eval, output;
set a %B001, set b %B000, eval, output;
// 由于芯片有两个3-bit位输入
// 所以完整测试就有2^3*3^2=64种组合
```

图 B.1　在硬件仿真器上测试芯片

set a4 -1，这些命令为四个变量赋予了相同的值：即 16 个 1 代表十进制中的“－1”。字符串值（前缀为%S）只能用于打印输出，而不能被赋予变量。字符串的内容必须包含在双引号（″ ″）中。

模拟时钟（仅用于时序芯片的测试）会发出一连串数值，用 0，0+，1，1+，2，2+，3，3+，等等来表示。这些**时钟周期**（*clock cycle*，也称为**时间单元**，*time unit*）的进程由 tick 和 tock 这两个脚本命令来控制。tick 命令将时钟的值从 *t* 推移到 *t*+，而 tock 命令将时钟的值从 *t*+推移到 *t*+1（即进入下一个时钟周期）。当前时钟周期被保存在称为 time 的系统变量中。

脚本命令可以访问三种类型的变量：管脚；内置芯片变量；系统变量 time 。

管脚：被模拟的芯片的输入管脚、输出管脚和内部管脚。例如，命令 set in 0 将 in 管脚的值设为 0。

Time：自模拟进程开始后流逝的时间单元的数量（只读）。

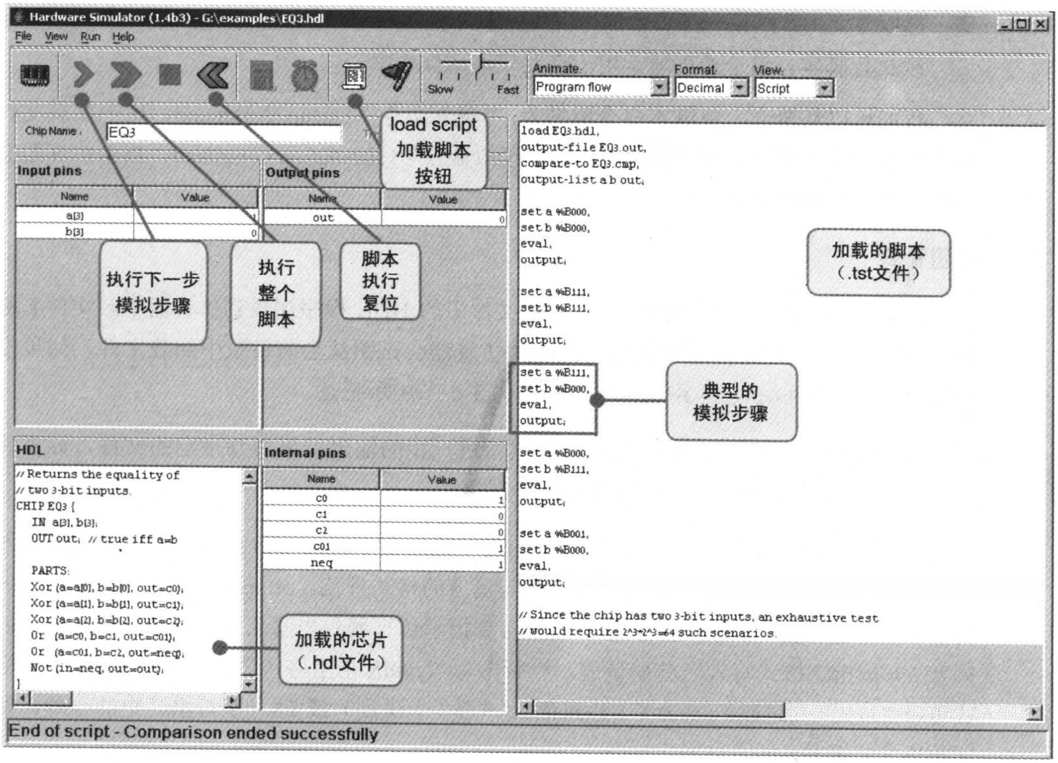


图 B.2 典型的硬件模拟过程，显示在脚本的尾部。除了增加一些加强可读性的空格，该脚本与图 B.1 中的 `EQ3.tst` 相同

B.2.3 脚本命令

Script Commands

命令语法 脚本就是命令的序列。每个命令以逗号、分号或感叹号结尾。这些标点符号的语意如下：

- 逗号（,）：结束一条脚本命令。
- 分号（;）：结束一条脚本命令和一个模拟步骤。模拟步骤可能由一条或多条脚本命令组成。当用户通过仿真器的 GUI 来指示仿真器"单步模拟"（single-step）时，仿真器会从当前命令开始执行，直到下一个分号为止，并且整个模拟进程会在该点暂停。

- 感叹号（!）：结束一条脚本命令并终止脚本的执行。当然，经后用户还可以从该点继续执行脚本。这个选项通常用于交互式调试操作中。

我们可以从概念上将脚本命令概括为："设置命令"（setup command，用于加载文件以及对模拟过程全局设置的初始化）和"模拟命令"（simulation command，指示仿真器去执行一系列的测试步骤）。

设置命令

load Xxx.hdl ：将存储在 Xxx.hdl 文件中的 HDL 程序加载到仿真器中。文件名称必须包含.hdl 扩展名，但不能包含路径。仿真器会试图从当前目录中加载文件，如果没有找到，则转到 builtIn 路径中寻找，如 A.3 小节中所描述。

output-file Xxx.out ：指示仿真器将进一步的输出结果写到指定的文件，该文件必须包含.out 扩展名，它会在当前路径中创建。

output-list v1,v2,... ：脚本中的每个输出命令将指示仿真器在输出文件里写入什么内容。列表中的每个值都是变量名及其对应的格式规范。output-list 中每个条目 v 的语法变量格式为：*padL.len.padR*。这条命令指示仿真器首先写入 *padL* 个空格，然后以列宽 *len* 按指定格式写入当前变量值，然后再写入 *padR* 个空格，最后写入分隔符"|"。格式可以是%B（二进制）、%X（十六进制）、%D（十进制）或%S（字符串）中的一种。默认的格式规范是%B1.1.1 。

例如，Hack 平台的 CPU.hdl 芯片有一个 reset 输入管脚、一个 pc 输出管脚和一个 DRegister 单元。如果我们希望在芯片的工作过程中跟踪这些变量的数值，可以使用以下命令：

```
output-list  time%S1.5.1            //系统变量
             reset%B2.1.2           //芯片的输入管脚
             pc%D2.3.1              //芯片的输出管脚
             DRegister[ ]%X3.4.4    //该内置单元的状态
```

该命令会产生下列输出（在两个后续的输出命令之后）：

```
| time | reset | pc | DRegister[] |
| 20+ |  0  | 21 |  FFFF    |
| 21  |  0  | 22|  FFFF    |
```

compare-to Xxx.cmp：指示仿真器将每个输出行与比较文件（必须包含.cmp 扩展名）中对应的行进行比较。如果发现两文件中任意两行不同，那么仿真器会显示错误消息并停止脚本执行。比较文件存在于当前目录下。

模拟命令

set *variable value* ：给变量赋值。该变量可以是芯片的管脚、内部管脚、内部单元。参与赋值操作的数值与变量之间的数据宽度应该匹配。例如，如果 x 是 16-比特位管脚，y 是 1-比特位管脚，那么 set x 153 是正确操作，而 set y 153 将产生错误并导致模拟进程终止。

eval ：指示仿真器应用芯片逻辑对当前输入管脚上的数值进行计算，并给出结果。

output ：该命令指示仿真器按以下步骤操作：

1. 获取所有变量的当前值，这些变量由最后一条 output-list 命令列出。
2. 按照最后一条 output-list 命令中所指定的格式产生一个输出行。
3. 将输出行写到输出文件。
4. （如果先前通过 compare-to 命令声明了一个比较文件）：如果输出行与比较文件的当前行不一样，就显示错误消息并停止脚本执行。
5. 分别将输出文件和比较文件的光标向前移一行。

tick ：结束当前时钟周期（时间单元）的第一阶段。

tock ：结束当前时钟周期的第二阶段，并开始下一个时钟周期的第一阶段。

repeat *num* {*commands*} ：该命令指示仿真器将包含在花括号中的命令重复执行 *num* 次。如果省略 *num*，仿真器会一直重复执行这些命令，直到因为某些原因导致模拟进程终止。

while *Boolean condition* {*commands*}：指示仿真器重复执行包含在花括号中的命令，直到布尔条件为真（true）为止。该布尔条件的格式是：*x op y*，*x* 和 *y* 是常数或变量，*op* 是=、>、<、>=、<=、<>操作中的一种。如果 *x* 和 *y* 是字符串，*op* 就是=或<>操作。

echo *text* ：指示仿真器在状态栏（仿真器 GUI 的一部分）中显示文本。该文本必须包含在双引号（" "）中。

clear-echo ：指示仿真器清空状态栏。

breakpoint *variable value* ：指示仿真器每执行一条脚本命令之后就将指定的变量值与特定数值进行比较。如果比较结果相等，就将脚本的执行挂起并给出提示消息。否则继续执行。

clear-breakpoint ：清除所有先前定义的断点。

built-in-chip *method argument(s)* ：内置芯片若是通过外部实现来替代，则外部实现就会暴露一些包含该芯片专用操作的方法。调用这些方法的语法随着内置芯片的不同而不同，这一点在下面会介绍。

B.2.4 内置芯片的方法和变量
Variables and Methods of Built-In Chips

芯片的逻辑可以通过 HDL 程序或高级编程语言来实现，前者称为“内置”，后者称为“外部实现”。通过语法 *chip Name*[*varName*]（其中 *varName* 是与外部实现相关的变量，并应该在芯片 API 有文档说明），内置芯片的外部实现可以方便地对芯片状态进行访问。图 B.3 中给出了本书所有内置芯片的 API（作为 Hack 计算机平台的一部分）。

例如，我们考虑命令 `set RAM16K[1017] 15` 。如果 RAM16K 是当前被模拟的芯片或是其内部单元，那么该命令会将 15 的 2-补码存入地址为 1017 的内存单元中去。又因为内置 RAM16K 芯片具有 GUI 功能，新存入的值会显示在芯片的图像中。

我们可以通过 *chipName*[ ]命令来访问内置芯片中某单一变量(single-value)的当前值。还可以用 *chipName*[*i*]命令访问内置芯片中的向量(vector)。例如，当模拟内置芯片 Register

芯片名	外在变量	数据类型/范 围	方 法
Register	Register[]	16-bit (−32768...32767)	
ARegister	ARegister[]	16-bit	
DRegister	DRegister[]	16-bit	
PC	PC[]	15-bit (0..32767)	
RAM8	RAM8[0..7]	每个数据项都是 16-bit 位	
RAM64	RAM64[0..63]	″	
RAM512	RAM512[0..511]	″	
RAM4K	RAM4K[0..4095]	″	
RAM16K	RAM16K[0..16383]	″	
ROM32K	RAM32K[0..32767]	″	Load Xxx.hack/Xxx.asm
Screen	Screen[0..16383]	″	
Keyboard	Keyboard[]	16-bit，只读	

图 B.3 本书提供的所有内置芯片的 API

时，可以通过脚本命令 `set Register[ ] 135` 将 Register 的值设置为 135 的 2-补码（该命令在下一时钟周期开始生效，并且 Register 开始对外输出 135）。

内置芯片还会暴露与实现相关的某些**方法**（*methods*），即那些“能够扩展仿真器命令功能”的方法。例如 Hack 计算机，程序驻留在指令内存中，该内存由 ROM32K 芯片实现。在该计算机上运行机器语言程序之前，必须首先将该程序加载到该 ROM32K 中。为了便于执行这个操作，ROM32K 的内置实现机制中有个 **`load`** *file name* 的方法，它来引用包含机器语言指令的文本文件。该方法可以通过诸如 `ROM32K load Myprog.hack` 的测试脚本命令来调用。在本书提供的芯片组中，该方法是唯一被所有内置芯片都支持的。

B.2.5 范例
Ending Example

本节最后介绍一个相对复杂的测试脚本，用于测试 Hack 平台的 Computer 芯片。一种测试方法是，将某个用机器语言编写的程序加载到该芯片，在计算机执行该程序的过程中，每执行一条指令，都对一些指定变量的值进行观测（看它们能否客观的反映出计算机运行过程中的状态）。例如，我们编写一个程序，用于选择 RAM[0]和 RAM[1]中的较大值，并将结果写入 RAM[2]中。该程序保存在文本文件 `Max.hack` 中。注意，我们是在计算机的

底层进行操作，如果该程序运行有误，可能是程序或硬件有误（若考虑更周全的话，也可能测试脚本或硬件仿真器有错误）。为了简单起见，我们假设：除被测试的 Computer 芯片之外，其他测试环境都是准确无误的。

为了利用 `Max.hack` 程序测试 Computer 芯片，我们编写了 `ComputerMax.tst` 测试脚本。该脚本将 `Computer.hdl` 加载到硬件仿真器中，然后将 `Max.hack` 程序加载到 Computer 芯片的 ROM32K 中。对该芯片合理的测试方法如下：将两个值分别放入 RAM[0]和 RAM[1]中，重启仿真器中的计算机，运行时钟，然后观察 RAM[2]。以上方法内部实现细节就是图 B.4 中描述的脚本所要做的工作。

如何断定 14 个时钟周期足够执行该程序呢？可通过“尝试-出错”的渐近方式来做到。首先用较大的值进行尝试，然后通过观察计算机输出（或通过分析当前被加载的程序在运行期的行为）来不断调整之前设定的值，直到计算机的输出趋于稳定为止。

B.2.6 默认脚本
Defaul Script

仿真器的 GUI 按钮（single step，run，stop，reset）并不控制被测芯片，而是控制被加载的测试脚本（该脚本控制被测芯片的操作）。那么，在当用户将某芯片加载到仿真器之前，没有为其加载相应的测试脚本时，仿真器将如何操作呢？在这种情况下，仿真器会使用以下默认脚本：

```
//硬件仿真器的默认脚本
repeat {
  tick;
  tock;
}
```

B.3 在 CPU 仿真器中测试机器语言程序
Testing Machine Language Programs on the CPU Eulator

本书提供的 CPU 仿真器能对基于 Hack 计算机平台上的二进制程序（第 5 章介绍过）进行测试，并对其运行情况进行模拟。被测程序可由 Hack 代码或第 4 章介绍的汇编语言编写。如果是后者，那么作为“load program（加载程序）”操作的一部分，仿真器会在运行过程中，动态地将所加载的汇编代码翻译成对应的二进制代码。

```
/*ComputerMax.tst 脚本。
 max.hack 程序对 RAM[0]和 RAM[1]两者进行比较,
 然后将较大值写入 RAM[2]。*/

// 加载 Computer 芯片, 并开始启动模拟
load Computer.hdl,
output-file Computer.out,
compare-to ComputerMax.cmp,
output-list RAM16K[0] RAM16K[1] RAM16K[2];

// 将 Max.hack 程序加载到 ROM32K 芯片
ROM32K load Max.hack,
// 将自定义的测试值存入 RAM16K 芯片的两个起始单元
set RAM16K[0] 3,
set RAM16K[1] 5,
output;
// 运行足够的时钟周期以完成程序的执行
repeat 14 {
   tick, tock,
   output;
}

// 复位 Computer
set reset 1,
tick,            // 开启时钟来执行程序
tock,            // Counter (PC, 时序芯片) 被重置为新的值
output;
// 现在以不同的测试值再次执行程序
set reset 0,     // 将复位取消 (下个时钟周期开始生效)
set RAM16K[0] 23456,
set RAM16K[1] 12345,
output;
repeat 14 {
   tick, tock,
   output;
}
```

图 B.4 测试最高层的抽象 Computer 芯片

用于测试机器语言程序（Xxx.hack 或 Xxx.asm）的脚本称为 Xxx.tst 。通常，整个模拟过程涉及 4 个文件：测试脚本（Xxx.tst），被测程序（Xxx.hack 或 Xxx.asm），可选的输出文件（Xxx.out），以及可选的比较文件（Xxx.cmp）。这些文件都必须保存在相同的目录下。方便起见，我们可将该目录命名为 xxx 。更多关于文件结构和用法的信息请参看 B.1 小节。

B.3.1 范例
Example

下面考虑乘法程序 Mult.hack ，它执行 RAM[2]=RAM[0]*RAM[1]的操作。测试该程序的合理方法之一是，将两个数值分别放入 RAM[0]和 RAM[1]中，在程序运行程序时，观察 RAM[2]中的数值。图 B.5 列出了该测试程序。

```
// 加载程序，开始启动模拟
load Mult.hack,
output-file Mult.out,
compare-to Mult.cmp,
output-list RAM[2]%D2.6.2;

// 将自定义的测试值存入 RAM16K 芯片起始的头两个单元中
set RAM[0] 2,
set RAM[1] 5;
// 运行足够的时钟周期以完成程序的执行
repeat 20 {
    ticktock;
}
output;

// 以不同的测试值重新执行同一个程序
set PC 0,
set RAM[0] 8,
set RAM[1] 7;
repeat 50 {          // 因为 Mult.hack 是基于重复的加法运算,
    ticktock;        // 所以被乘数越大，所需的时钟周期就越多
}
output;
```

图 B.5 在 CPU 仿真器上测试机器语言程序

B.3.2 变量
Variables

CPU 仿真器与硬件细节相关（hardware-specific），它能辨认一组与 Hack 平台的内部组件相关的变量。特别是，运行在 CPU 仿真器上的脚本命令可以访问下列元素：

A：地址寄存器的值（无符号 15-比特位）；

D：数据寄存器的值（16-比特位）；

PC：程序计数寄存器的值（无符号 15-比特位）；

RAM[i]：内存单元 RAM[*i*]中存储的数值（16-比特位）；

time：自模拟开始后所流逝的时钟周期的个数（只读）。

B.3.3 命令
Commands

CPU 仿真器几乎能够支持 B.2.3 小节中介绍的所有命令，但是有如下变化：

load *program*：这里 *program* 是 `Xxx.hack` 或 `Xxx.asm`。该命令将（待测试的）机器语言程序加载到模拟的指令内存中。如果该程序是用汇编语言编写的，它将在模拟过程中被翻译成二进制码。

eval：不适用。

built-in-chip *method argumen*：不适用。

ticktock：该命令用来替代 `tick` 和 `tock`。每个 `ticktock` 命令使时钟前进一个时间单元。

B.3.4 默认脚本
Default Script

仿真器的 GUI 按钮（single step，run，stop，reset）并不控制被加载的芯片，而是控制被加载的脚本（该脚本控制被加载芯片的操作）。如果当用户将芯片加载到仿真器之前没有加载对应的脚本文件时，仿真器将如何操作呢？在这种情况下，仿真器会使用下面的默认脚本：

```
// CPU 仿真器的默认脚本
repeat {
   ticktock;
}
```

B.4 在 VM 仿真器中测试 VM 程序 Testing VM Programs on the VM Emulator

第 7～8 章中介绍了虚拟机（VM）的模型和 Hack 平台上的 VM。我们提供的 VM 仿真器是用 Java 编写的，能运行 VM 程序并将其操作可视化，显示相关虚拟内存段的状态。

前面介绍过，VM 程序由一个或多个`.vm`文件组成。因此，对 VM 程序的模拟将包含 4 个部分：测试脚本（`Xxx.tst`），被测的程序（单一的`.vm`文件或一个包含若干`.vm`文件的`xxx`目录），一个可选的输出文件（`Xxx.out`），以及一个可选的比较文件（`Xxx.cmp`）。所有这些文件必须保存在相同路径下，方便起见，可以将该目录命名为`xxx`。更多关于文件结构和用法的信息请参看 B.1 小节。第 7 章提供了虚拟机结构的背景知识，下面的讨论基于这些背景知识展开。

启动代码 VM 程序通常应至少包含两个函数：`Main.main`和`Sys.int`。当 VM 翻译器翻译 VM 程序时，它会生成对应的机器代码，该代码将堆栈指针设置为 256，然后调用`Sys.init`函数，`Sys.init`函数会继续调用`Main.main`。与此类似，当 VM 仿真器去执行 VM 程序（由一个或多个 VM 函数组成的集合）时，它首先会运行`Sys.init`函数（该函数应该在被加载的 VM 代码中），如果没有找到该函数，那么仿真器就开始执行 VM 代码中的第一条命令。

为了有利于对 VM 进行逐步的开发（因为 VM 开发涉及到两个章节，开发被分为两个阶段），我们的 VM 仿真器采用了上述的后一种方案。在第 7 章里，我们仅构建了用于处理`pop`、`push`和运算命令的部分 VM 功能，而没有实现子程序调用命令的相关功能。因此，针对项目 7 的测试程序都是由“原始的（raw）”VM 命令组成的，并不具有典型的`function/return`形式的嵌套展开处理。为了能让用户对这些命令做非常规试验，我们让 VM 仿真器具有执行“原始的（raw）”VM 代码的能力，这些代码既没有被正常初始化，也没有被封装在函数结构中。

虚拟内存段 在模拟虚拟机操作的过程中，VM 仿真器负责管理 Hack VM 的虚拟内存段（`argument`、`local`等）。这些段必须在宿主 RAM 中分配，因为仿真器对一个任务

的执行与通过 `call`、`function` 和 `return` 命令进行函数调用的过程相似。这意味着即使在模拟那些不包括程序调用命令的“原始”VM 代码时，我们也必须强制 VM 仿真器去显式地在宿主 RAM 中分配虚拟内存段（至少是当前代码中涉及的那些段）。这个初始化操作可以通过一些特定的脚本命令完成，这些脚本命令能够操作指向各虚拟内存段基地址的指针。这样，我们可以将虚拟段映射到宿主 RAM 中指定的区域。

B.4.1 范例
Example

`FibonacciSeries.vm` 文件包含一系列计算 Fibonacci 序列中前 *n* 个元素的 VM 命令。该代码有两个参数：数值 *n* 和用于存放操作数的起始内存地址。图 B.6 中的脚本用实际参数 6 和 4000 来测试该程序。

B.4.2 变量
Variables

运行在 VM 仿真器上的脚本命令可以访问下列元素：

虚拟内存段的内容

`local[i]`：`local` 段第 i 个元素的值；

`argument[i]`：`argument` 段第 i 个元素的值；

`this[i]`：`this` 段第 i 个元素的值；

`that[i]`：`that` 段第 i 个元素的值；

`temp[i]`：`temp` 段第 i 个元素的值。

虚拟内存段的指针

`local`：`local` 段在 RAM 中的基地址；

`argument`：`argument` 段在 RAM 中的基地址；

`this`：`this` 段在 RAM 中的基地址；

`that`：`that` 段在 RAM 中的基地址；

```
/*FibonacciSeriers.vm 文件包含一系列 VM 命令,
  这些命令用来计算 Fibonacci 数列的前 n 个数。
  该程序的代码不包含 function/call/return 命令,
  因此必须要求 VM 仿真器初始化那些将被代码显式使用的虚拟内存段。*/
//加载程序，开始启动模拟
load FibonacciSeries.vm,
output-file FibonacciSeries.out,
compare-to FibonacciSeries.cmp,
output-list RAM[4000]%D1.6.2 RAM[4001]%D1.6.2 RAM[4002]%D1.6.2
            RAM[4003]%D1.6.2 RAM[4004]%D1.6.2 RAM[4005]%D1.6.2;
// 初始化 stack 段, argument 段以及 local 段。
set SP 256,              // 设置堆栈指针(堆栈起始地址为 RAM[256])
set local 300,           // 将 local 段的基地址设为某个 RAM 地址
set argument 400;        // 将 argument 段的基地址设为某个 RAM 地址
// 将两个测试值放入 arguments 段中的两个单元
set argument[0] 6,       // n=6
set argument[1] 4000;    // 将序列放在从 RAM[4000] 起始的位置
// 执行足够的 VM 步骤以完成程序的执行
repeat 140 {
  vmstep;
}
output;
```

图 B.6　在 VM 仿真器上测试 VM 程序

与实现相关的变量

RAM[i] ：地址为 i 的内存单元中存储的数值；

SP ：堆栈指针的值；

currentFunction ：当前执行函数的名称（只读）；

line ：包含一个形如 *current-function-name.line-index-in-function* 的字符串（只读）。

例如，当程序执行到函数 `Sys.init` 的第 3 行时，`line` 的内容是“`Sys.init.3`”，它对在 VM 程序中指定位置的断点设置是很有用的。

B.4.3 命令
Commands

VM 仿真器除了以下几点变化外，能够支持 B.2.3 节中介绍的所有命令：

load *source*：其中 *source* 是包含一个或多个 VM 函数的 Xxx.vm 文件名，或是一个 VM 命令序列，或是包含若干.vm 文件的目录名（假设所有.vm 文件都被加载）。

如果.vm 文件位于当前目录下，那么参数 *source* 可以省略掉。

tick/tock ：不适用。

vmstep ：模拟 VM 程序中 VM 命令的执行，并为执行代码中下一条命令作准备。

B.4.4 默认脚本
Default Script

仿真器的 GUI 按钮（single step，run，stop，reset）并不控制被加载的芯片，而是控制被加载的脚本（该脚本控制被加载芯片的操作）。如果当用户将一个芯片加载到仿真器之前没有加载对应的脚本文件时，仿真器将如何操作呢？在这种情况下，仿真器会使用下面的默认脚本：

```
//CPU 仿真器的默认脚本
repeat {
  vmstep;
}
```

索引

Abstraction, 6, 263
implementation paradigm, xi–xii
modules and, 2–3
Adder gates, 29–39
Addresses, 45, 104
direct addressing, 60–61
indirect addressing, 61
machine language and, 60–61, 63
mapping and, 84–91, 137–141
memory and, 81–82 (*see also* Memory)
program size limits and, 106
registers and, 45, 83–86
subroutines and, 153–159
symbol table and, 105
VM-Hack mapping and, 139–143, 161–168
Addressing instruction (*A*-instruction), 64–65, 108–110, 115
Algorithms
efficiency and, 249, 272–273
graphics and, 257–263
mathematics and, 248–252
memory management and, 252–256
operating systems and, 272–273 (*see also* Operating systems)
runtime and, 249
syntax and, 250
ALU. *See* Arithmetic Logic Unit
Analysis-synthesis paradigm, 223
And function, 8–9, 20
implementation of, 26
multi-bit versions of, 21–23
Application Program Interface (API) notation, 19
Architecture, x, 79, 99–101
bottom-up, 3–4
chip set, 2
CPU and, 82–83
Hack, 5–6, 85–98
hardware, 2
I/O and, 84–85
Jack, 175–176 (*see also* Jack)
machine language and, 106–107
memory and, 81–82
modifications and, 277–279
modules and, 2–3
optimization and, 80
registers and, 83–84
sequential chip hierarchy and, 47–50
standards and, 84
stored program concept and, 80
top-down, 3
VM and, 121–151 (*see also* Virtual Machine)
von Neumann, 62, 79–81, 85
Aristotle, 6
Arithmetic addition, 37
Arithmetic Logic Unit (ALU), 2, 6, 39
Boolean arithmetic and, 29, 32, 35–38
combinational chips and, 46–47
CPU and, x, 82–83, 94
description of, 35–38
Hack and, 86

Arithmetic Logic Unit (ALU) (cont.)
operating systems and, 248–249
visualized chip operations and, 292
Arrays, 81
data translation and, 224–231
Jack and, 175, 184–185, 191, 265, 269
operating systems and, 256, 265, 269
stack processing and, 124–127
variable-length, 256
Virtual Machine (VM) and, 137
ASCII code, 71, 89, 252
Assembler, x, 5, 71–72, 75–76, 118–120, 277
hash table, 115
implementation of, 112–116
labels and, 105
machine language specification and, 107
macros and, 117
mnemonics and, 108, 114
program size limits and, 106
symbols and, 60, 104–106, 110–111, 114–116, 143, 164
syntax and, 104, 107–110
test scripts and, 103–104
as translator program, 104–107, 163–164
variables and, 105

Best-fit, 254
Big-Oh notation (*O*(n)), 249
Binary code, 5, 108. *See also* Boolean logic
code generation and, 223–246
graphics and, 257–263
Jack and, 174
Binary search, 251
Bitmaps, 259–263, 269
Bit shifting, 60
Bit-wise negation, 60
Boolean arithmetic, x
addition, 30
algebra and, 8–10
ALU and, 29, 32, 35–38
binary numbers and, 30–32
CPU and, 29
least significant bits (LSB), 30
memory and, 42–47
most significant bits (MSB), 30
radix complement method, 31
signed binary numbers, 31–32
stack processing and, 126–127
Boolean logic
abstraction of, 11
algebra and, 8–10
canonical representation, 9
conditional execution, 62
gates and, 8, 11–13
hardware construction and, 13–14
HDL and, 14–17
machine language and, 57–77
repetition, 61–62
subroutine calling, 62
truth tables, 8
two-input functions, 9–10
Bootstrap code, 165
Buses, 21–22, 286–287

C#, 4–5, 112, 121, 147, 169
Jack and, 174, 196
C++, 112, 147, 253
Canonical representation, 9
Case conventions, 108
Central Processing Unit (CPU), 6, 29, 59
ALU and, 82–83, 94
architecture and, 82–83
control unit and, 82–83
description of, 82–83
emulators and, 306–309
Hack and, 62–63, 85–96
instruction memory and, 82
program counter and, 84
registers and, 82
testing and, 306–309
von Neumann architecture and, 81
Character output, 259–263, 269
Chips, 2. *See also* Gates
adder, 29–39
API specification and, 19
Boolean logic and, 7–28
built-in, 287–288, 293, 296, 304–305
buses and, 286–287
clocks and, 289–291
combinational, 41, 46–47

connections and, 285–286
cost and, 14–15
description of, 11
efficiency and, 288
feedback loops and, 291–292
Hack platform and, 85–91
hardware simulator and, 283–284
HDL and, 14–17, 281–296
incrementer, 33–39
maintaining state and, 41–42
pins and, 284–286
RAM, 86
ROM, 85
sequential, 41–55, 289–292
simulators and, 299–306
testing and, 297–313
visualized operations for, 288, 292–296
Clocks, 41, 48, 289–290
feedback loops and, 291–292
memory and, 42, 52–54
Code generation
commands translation and, 231–232
data translation and, 224–231
operating systems and, 272 (*see also* Operating systems)
registers and, 223–224
syntax analysis and, 237–241
virtual machines and, 224
Combinational logic. *See* Boolean arithmetic
Commands translation, 231–232
Common Language Runtime (CLR), 123, 146–147
Communications, 279
Compare file, 18
Compilers, ix–x, 2, 5–6, 17, 103, 112
abstraction and, 175–179
analysis-synthesis paradigm and, 223
code generation and, 223–246
description of, 199–201
grammars and, 203, 206–207
Hack and, 133–134 (*see also* Hack)
high-level language and, 146–147
Jack and, 133–134, 174, 193–195 (*see also* Jack)
lexical analysis and, 202, 208
mapping and, 137–141
memory allocation and, 234
nested subroutine calling and, 153
parsing and, 203–207
p-code and, 123, 146
semantics and, 199
syntax analysis and, 199–221, 237–241
VM and, 122–127, 161–168, 233–235 (*see also* Virtual Machine)
XML and, 199–201, 211–218, 221
Complex Instruction Set Computing (CISC), 98
Composite gates, 11–13
Compute instruction (*C*-instruction), 66–69, 86, 108–110, 115
Computers. *See also* Architecture
ALU and, 29 (*see also* Arithmetic Logic Unit)
Boolean abstraction and, 11
bootstrap code and, 165
CPU and, 29 (*see also* Central Processing Unit)
dedicated, 97
emulators and, 121–122
general-purpose, 97–98
HDL and, 14–17 (*see also* Hardware Description Language)
machine language and, 57–77
memory and, 81–82
program flow and, 153–159
stored program concept and, 79–80
Conditional execution, 62
Conditional jump, 62
Constants, 181–182
Control logic, 94–95
Control unit, 82–83
Converters. *See* Not function
Counters, x, 45, 47–48, 50, 52, 84, 95
CPU. *See* Central Processing Unit
Cycles, 42

Data flip-flop (DFF)
clocked chips and, 290–291
implementation of, 50–51
sequential logic and, 42–48

connections and, 285–286
cost and, 14–15
description of, 11
efficiency and, 288
feedback loops and, 291–292
Hack platform and, 85–91
hardware simulator and, 283–284
HDL and, 14–17, 281–296
incrementer, 33–39
maintaining state and, 41–42
pins and, 284–286
RAM, 86
ROM, 85
sequential, 41–55, 289–292
simulators and, 299–306
testing and, 297–313
visualized operations for, 288, 292–296
Clocks, 41, 48, 289–290
feedback loops and, 291–292
memory and, 42, 52–54
Code generation
commands translation and, 231–232
data translation and, 224–231
operating systems and, 272 (*see also* Operating systems)
registers and, 223–224
syntax analysis and, 237–241
virtual machines and, 224
Combinational logic. *See* Boolean arithmetic
Commands translation, 231–232
Common Language Runtime (CLR), 123, 146–147
Communications, 279
Compare file, 18
Compilers, ix–x, 2, 5–6, 17, 103, 112
abstraction and, 175–179
analysis-synthesis paradigm and, 223
code generation and, 223–246
description of, 199–201
grammars and, 203, 206–207
Hack and, 133–134 (*see also* Hack)
high-level language and, 146–147
Jack and, 133–134, 174, 193–195 (*see also* Jack)
lexical analysis and, 202, 208
mapping and, 137–141
memory allocation and, 234
nested subroutine calling and, 153
parsing and, 203–207
p-code and, 123, 146
semantics and, 199
syntax analysis and, 199–221, 237–241
VM and, 122–127, 161–168, 233–235 (*see also* Virtual Machine)
XML and, 199–201, 211–218, 221
Complex Instruction Set Computing (CISC), 98
Composite gates, 11–13
Compute instruction (*C*-instruction), 66–69, 86, 108–110, 115
Computers. *See also* Architecture
ALU and, 29 (*see also* Arithmetic Logic Unit)
Boolean abstraction and, 11
bootstrap code and, 165
CPU and, 29 (*see also* Central Processing Unit)
dedicated, 97
emulators and, 121–122
general-purpose, 97–98
HDL and, 14–17 (*see also* Hardware Description Language)
machine language and, 57–77
memory and, 81–82
program flow and, 153–159
stored program concept and, 79–80
Conditional execution, 62
Conditional jump, 62
Constants, 181–182
Control logic, 94–95
Control unit, 82–83
Converters. *See* Not function
Counters, x, 45, 47–48, 50, 52, 84, 95
CPU. *See* Central Processing Unit
Cycles, 42

Data flip-flop (DFF)
clocked chips and, 290–291
implementation of, 50–51
sequential logic and, 42–48

Goto operation, 95, 155
Grammars
 Jack and, 203, 207–215
 parsing and, 203–207
 syntax analyzer and, 207–213
Graphical User Interface (GUI), 247, 283, 288–290
 testing and, 297–313
 visualized chip operations and, 292–296
Graphics, 98
 character output, 259–261
 circle drawing, 259
 keyboard handling and, 261–263
 line drawing, 257–258
 multiplication and, 258–259
 pixel drawing, 257
GUI. *See* Graphical User Interface

Hack, 5, 35, 79
 address instruction format and, 64–65, 85–86
 assembler, 75–76, 103–120
 built-in chips and, 293, 296
 case conventions and, 108
 case sensitivity, 75
 C-instruction, 66–69
 CPU and, 62–63, 76–77, 85–96
 destination specification and, 66–68
 file formats and, 71–72, 107–110
 graphics card and, 98
 input/output (I/O) handling and, 70–71, 98
 instructions and, 108–110
 Internet and, 279
 jump specification, 68–69
 memory and, 63, 87–91, 96
 modifications and, 278–279
 platform description, 62–64, 85–98
 symbols, 69–70
 syntax, 71–73, 107–110
 VM mapping and, 139–143, 161–168
Half-adder chip, 32–33, 38
Hardware, ix–x, 4–6. *See also* Input/output
 architecture of, 2, 79–101
 Boolean logic and, 8–28
 chips and, 85, 293–296 (*see also* Chips; Gates)
 keyboard, 71
 machine language and, 57–77
 memory and, 81–82
 modifications and, 278–279
 operating systems and, 247–276
 RAM, 42–47
 screen, 70
 sequential chips and, 41–55
 simulators and, 299–306
 stored program concept and, 79–80
Hardware Description Language (HDL), x–xiii, 5–6, 93, 278
 API notation and, 282
 case sensitivity and, 283
 chip logic and, 17–25, 281–296
 compare file, 18
 description of, 281
 efficiency and, 288
 hardware simulator and, 14, 17–25, 283–284
 header section, 15
 identifier naming and, 283
 interfaces and, 15–16
 logic building and, 39
 parts section, 15
 statement representation, 15
 technical references for, 281–296
 testing and, 16–17
 visualized chip operations and, 292–296
Hardware simulator, 14, 283–284
 chip specifications and, 17–25
Hash tables, 115, 226
HDL. *See* Hardware Description Language
Heap, 132–133
High-level language, 4–6
 Jack, 173 (*see also* Jack)
 operating systems and, 248
 program flow and, 153–159
 subroutines and, 62, 112, 153–161, 181–190, 195, 209, 234–235
 VM-Hack mapping and, 139–143, 161–168

If-goto destination, 155
If-*x*-then-*y* function, 10
Immediate addressing, 61
Incrementer chip, 33–39
Indirect addressing, 61
Inheritance, 195–196, 241–242
Input/output (I/O), x
characters and, 259–263
device driver, 256–257
graphics, 257–263
Hack and, 62–77, 70–71, 98
keyboards, 261–263, 266
operating systems and, 256–270
screens, 265–266
standards and, 84
Instructions, 116
addresses, 64–65, 108–110 (*see also* Addresses)
assembler and, 103–120
CISC, 98
compilers and, 122–127 (*see also* Compilers)
compute, 66–69, 108–110
decoding, 94–96
execution, 94–96
fetching, 86, 95–96, 98
labels and, 105
macros and, 117
memory and, 63, 82
RISC, 98
stack processing and, 130 (*see also* Stack processing)
subroutines and, 62, 112, 153–161, 181–190, 195, 209, 234–235
symbolic vs. binary, 104
variables and, 105
Interfaces, 282, 284
HDL and, 15–16
logic gates and, 12
Intermediate language (IL), 123
Internal pins, 15–16

Jack, 1, 4–5, 147, 165, 169, 197
abstract data types and, 175–179
API notation and, 175–176, 200, 215, 224
applications writing, 193–195
array handling, 175, 184–185, 191, 265, 269
binary code and, 174
classes and, 175–183, 187–193, 208, 248, 263–273
code generation and, 223–246
constants, 181–182
constructor for, 234–235
data types and, 183–185
evaluation order, 188
expression evaluation and, 187–188, 231–232
flow control and, 231–232
generic statements, 187
grammar and, 203, 207–215
identifiers, 181–182
inheritance and, 195–196, 241–242
I/O and, 191–193, 209–215, 265–266, 269–270
Java and, 174, 183, 196
keyboards and, 192–193, 266, 270
lexical analysis and, 202, 208
linked list implementation, 179–180
`Main.main` function, 174–175
memory and, 193, 266–267, 270–271
modifications and, 277–278
as object-based language, 173, 195–196, 199
object handling and, 189–190, 228–231
operating system, 195, 197, 235, 253, 257–273
operator priority, 188
parsing and, 200, 217, 221
program elements in, 133–134
rational numbers and, 175–179
reserved words, 181–182
screens and, 192, 265–266, 269
simplicity of, 174
standard library of, 174, 190–193, 196, 263
strings and, 191, 264–265, 268–269
subroutines and, 181–190, 195, 209, 234–235
symbols and, 181–182, 238–239
syntax and, 181–182, 187, 207–221, 237–241
tokenizing and, 181, 202, 205, 208, 214–215, 219–221

type conversions, 183, 241
variables and, 181–187
VM code and, 174, 233–235, 240
void methods and, 235
white space, 181–182
XML and, 199–201, 211–218, 221
Java, 17, 247, 253, 277
assembler and, 112
built-in chips and, 293, 296
Jack and, 174, 183, 196
stack arithmetic and, 122, 134
standard libraries, 147
VM and, 122, 134, 169
Java Runtime Environment, 123, 146
Java Virtual Machine (JVM), 121, 123, 146
Jump, 109–110, 114
nested subroutine calling and, 153–159
specification, 61–62, 68–69, 96

Keyboard input, 71, 84, 86, 89, 96
Jack and, 192–193, 266, 270
operating systems and, 266, 270
string reading and, 262–263
text handling and, 261–263
visualized chip operations and, 292–293

Labels, 70, 105, 110, 116, 155, 159
Last-in-first-out (LIFO) storage model, 124, 157
Least significant bits (LSB), 30
Lexical analysis, 202, 208
XML and, 199–201, 211–218, 221
Lexical analysis (LEX) tool, 217
Line drawing, 257–259
Linked list, 179–180
Linux, xiii, 277
Load command, 60
Logic
Boolean, 7–28 (*see also* Boolean logic)
control logic and, 94–95
decoding, 94–96
fetching, 95–96
HDL and, 281–296
instruction execution, 94–96
jumps, 61–62, 68–69, 96
stack processing and, 130 (*see also* Stack processing)
stored program concept and, 79–80

Machine language, x
abstraction and, 81
addressing and, 60–61, 63
assembler and, 103–120
binary codes and, 59–60
commands and, 60–62
compilers and, 122–127 (*see also* Compilers)
conditional execution, 62
Hack, 62–77
instruction memory and, 82
labels and, 105
memory and, 58–62
mnemonic symbols, 59
processor and, 59
program size limits and, 106
registers and, 59
repetition and, 61–62
subroutine calling, 62
symbolic vs. binary, 104
syntax and, 60–62, 71–73, 104
testing and, 306–309
unconditional jump, 62
variables and, 105
VM and, 122–127 (*see also* Virtual Machine)
Macro commands, 117
Mapping
I/O operations and, 84–91
keyboard handling and, 262–263
memory segments and, 142–143
VM-to-Hack, 139–143, 161–168
VM-to-Jack, 233–235
Memory, 2
addresses and, 45, 91 (*see also* Addresses)
allocation and, 253–254
arrays and, 227–228
clocks and, 42, 52–54
compilers and, 234
dynamic allocation and, 252–253
flip-flops and, 42–54

Memory (cont.)
 fragmentation and, 254, 256
 graphics and, 257–263
 Hack and, 63, 87–91, 96
 implementation and, 50–52
 improved allocation and, 254–256
 instruction, 63, 82
 Jack and, 193, 266–267, 270–271
 machine language and, 58–62
 mapped input/output (I/O) and, 84–91
 object handling and, 228–231
 operating systems and, 247, 252–256, 266–267, 270–271
 RAM, 42–45, 49–50 (*see also* Random access memory)
 registers and, 42–49
 stored program concept and, 79–80
 subroutines and, 62, 112, 153–161, 181–190, 195, 209, 234–235
 testing and, 310–311
 variable locations and, 106
 virtual segment mapping and, 142–143
 visualized chip operations and, 292
 VM and, 127–133
 von Neumann architecture and, 81
Mnemonics, 59, 108, 114
Multi-bit bus, 286–287
Multiplexors, x, 20–26
Multiplication, 249–250, 258–259
Multitasking, 247

Nand function, 2, 7, 10, 19, 27
Negative numbers, 31–32
Nested subroutine calling, 153–159
.NET infrastructure, 122, 123, 146–147
Network interface cards, 84
Newton-Raphson method, 251
Non-terminals, 203, 211
Nor function, 2, 10
Not function, 8–9, 26
Number base, 30

Object types, 183–184
Operating systems, ix–x, 4
 API notation and, 263, 267
 arrays and, 256, 265, 269
 classes and, 264–271
 description of, 247
 graphics and, 257–263
 hardware/software gaps and, 247
 initialization and, 267
 input/output (I/O) management, 256–266, 269–270
 Jack and, 195, 235, 263–273 (*see also* Jack)
 mathematical operations and, 248–252, 264, 268
 memory and, 247, 252–256, 266–267, 270–271
 program size limits and, 106
 screens and, 265–266, 269–270
 strings and, 252, 256, 264–265, 268–269
 Sys and, 267, 271
Operator priority, 188
Or function, 8–9, 20
 implementation of, 26
 multi-bit versions of, 21–23
 multi-way versions and, 23–25
Overflow, 30

Parsing, 2, 17, 60, 116
 assembler and, 112–114
 compilers and, 217 (*see also* Compilers)
 expression evaluation and, 187–188, 231–232
 grammar and, 203–207
 Jack and, 200, 203–207, 217, 221
 programming and, 107
 recursive descent, 204–206
 symbol-less, 114–115
 VM and, 144–146, 168
Pascal, 123, 146
P-code, 123, 146
Pins, 11, 15–16, 284–286, 290, 300
Pixel drawing, 257
Pointers, 69–70, 124, 131, 142, 161
Pop operation, 124, 130–132
Positive numbers, 31–32
Postfix notation, 231–232
Primitive gates, 11–13, 25–26
Program counter, 45, 84, 95

Program flow
assembly language symbols and, 164
bootstrap code and, 165
calling protocol and, 160–161
LIFO model and, 157
nested subroutine calling and, 153–159
VM, 129–130, 133–134, 153–168
Push operation, 124, 130–132

Radix complement method, 31
RAM. *See* Random access memory
Random access memory (RAM), x, 6, 278–279
clocked chips and, 290–291
Hack platform and, 86, 96, 139–143, 161–168
implementation of, 52
memory management and, 253
operating systems and, 270–271
registers and, 49–50
sequential logic and, 42–47
testing and, 304–308, 311–312
VM and, 137–143, 161–168
Rational numbers, 175–179
Read-only memory (ROM) chips, 6, 85–86, 91, 278–279
Read/write operations
memory and, 42–47
registers and, 48–49
Recursive descent parsing, 204–206
Reduced Instruction Set Computing (RISC), 98
Registers, x, 2
addresses and, 45, 83–86
API specification and, 48–49
architecture of, 83–84
CPU and, 82
Hack and, 63–64, 69
implementation of, 52
machine language and, 59
memory and, 42–49
RAM and, 49–50
read-write operations and, 48–49
testing and, 304–305
virtual, 69
visualized chip operations and, 292
Reserved words, 181–182
Return address, 158
Right Polish Notation (RPN), 231–232
Rogers, Carl, 1
RPN. *See* Right Polish Notation

Screen output, 70, 84, 86, 89, 96
characters and, 259–263
graphics and, 257–263
Jack and, 192, 265–266, 269
operating systems and, 265–266, 269–270
resolution and, 257–258
visualized chip operations and, 292–296
Segment index, 131–132, 135
Selectors, 20
Semantics, 199. *See also* Symbols; Syntax
data translation and, 224–231
Sequential logic, 6
chip hierarchy and, 47–50
clocks and, 289–291
feedback loops and, 291–292
flip-flops and, 41–54
memory and, 42–47
time and, 45–47
Signed binary numbers, 31–32
Simulators, 101
testing and, 297, 299–306
Square root function, 251
Stack pointer, 124, 131, 142, 161
Stack processing, 122
arithmetic and, 126–130
bootstrap code and, 165
heap structure and, 132–133
LIFO model and, 124, 157
memory and, 130–133
model of, 124–127
nested subroutine calling and, 153–159
pop operation, 124, 130–132
push operation, 124, 130–132
subroutines and, 153–159
VM-Hack mapping and, 139–143, 161–168
Standard language library, 4

Standard mapping, 141
Store command, 60
Stored program concept, 79–80
Strings, 184
 Jack and, 191, 264–265, 268–269
 keyboard handling and, 262–263
 operating systems and, 252, 256, 264–265, 268–269
Subroutines, 62, 112
 calling protocol and, 160–161
 code generation and, 234–235
 functional commands and, 153–159
 Jack and, 181–190, 195, 209, 234–235
 LIFO model and, 157
 void, 235
Switching technology, 2, 11
Symbols
 assembler and, 60, 104–106, 110–111, 114–116, 143, 164
 function calling and, 160
 Jack and, 181–182
 labels, 70, 105, 110, 116, 155, 159
 machine language and, 59–60, 69–70, 104
 mnemomic, 59
 resolution and, 105–106
 variables and, 105
Symbol tables, 103, 105, 115–116, 243
 data translation and, 225–226
 Jack and, 238–239
Syntax, 5, 104
 expression evaluation and, 187–188, 231–232
 formal languages and, 201–202
 non-terminals and, 203, 211
 RPN, 231–232
 semantics and, 199
 terminals and, 203, 211
 testing and, 301–304
 XML and, 199–201, 211–218, 221

Taylor series, 251
Terminals, 203, 211
Testing
 chips, 299–306
 emulators and, 297, 306–313
 GUI and, 297–298
 machine language and, 306–309
 script commands and, 301–304
 simulators and, 297, 299–306
 test scripts, 16–17, 103–104
 VM and, 310–313
Text files, 2
Time
 clocks, 41–54, 289–292
 counters, x, 45, 47–48, 50, 52, 84
 sequential logic, 6, 42–54, 289–292
 testing and, 297–313
Tokens, 181
 Jack tokenizing, 202, 205, 208, 214–215, 219–221, 237–241
 syntax analyzer and, 207–213
Transistors, 2, 11
Translator program, 163–164
Truth tables, 8–9
Turing, Alan, 122
Turing machine, 80–81
Two-input Boolean functions, 9–10
2's complement method, 30

Unconditional jump, 62
Unix, 272

Variables, 105, 116
 argument, 234
 fields, 183, 185, 226, 227
 Jack and, 181–187
 local, 183, 185–186, 227, 234, 253
 parameter, 183, 185–186
 scope and, 1, 225–226
 static, 183, 185, 226–227, 234, 253
Virtual Hardware Description Language (VHDL), 14
Virtual Machine (VM)
 advantages of, 121, 123–124
 arithmetic and, 126–130, 135
 array handling and, 137
 bootstrap code and, 165
 class and, 129
 compilers and, 122–127
 design suggestions for, 143

emulators and, 121–122, 150–151
examples of, 135–139
functions and, 127, 129–130, 133, 135–139
Hack mapping and, 139–143, 161–168
heap structure and, 132–133
high-level language and, 146–147, 153–154
implementation, 55, 103, 112
Jack and, 174, 233–235, 240
language for, 122
memory and, 127, 129–133
modifications and, 277–279
modularity and, 123–124
nested subroutine calling and, 153–159
object handling and, 137–139
program flow and, 129–130, 133–134, 153–168
stack processing and, 124–127
subroutines and, 154–159
symbols and, 143
syntax and, 123
testing and, 310–313
translator, 121
Virtual memory segments, 131
Visual Basic, 147
VM. *See* Virtual Machine
Void methods, 235
von Neumann architecture, 62, 79–81, 85

White space, 108, 113, 181–182
Windows, xiii, 277
Working stack, 161

XML, 199–201, 211–218, 221
Xor function, 10, 20
implementation of, 26
multi-bit versions of, 21–23

Yet Another Compiler Compiler (YACC), 217

Broadview
WWW.BROADVIEW.COM.CN
技术凝聚实力 专业创新出版

计算机系统要素

从零开始构建现代计算机

在计算机科学发展的早期，硬件、软件、编译器，以及操作系统之间的交互是非常简单的，人们很容易掌握整个计算机的工作原理。然而随着计算机技术的日益复杂，这种明晰性不复存在。与其他一些涵盖计算机科学某个具体领域知识的书不同，本书为读者呈现的是一幅完整且严格的应用计算机科学的画卷，它指导读者构建一个简单，然而功能强大的计算机系统。

事实上，理解计算机工作原理的最好方法就是亲自动手，从零开始构建计算机系统。本书通过12个章节和项目来引领读者从头开始，逐步地构建一个基本的硬件平台和现代软件层级。在这个过程中，读者能够获得实际的关于硬件体系结构、操作系统、编程语言、编译器、数据结构、算法，以及软件工程的知识。通过这种构造化的方法，本书揭示了计算机科学知识中的一个重要部分，并且论证了其他课程中所介绍的理论和应用技术是如何融入这幅画卷中的。

本书可以作为一个学期或两个学期的课程讲授。基于抽象实现的模式，每一章都介绍一个关键的硬件或软件抽象、一种实现方式和一个实际的项目。这个计算机系统可以按照章节介绍的顺序来实现，当然这只是一种选择，因为这些项目之间是彼此独立的，我们可以按照任意的顺序来实现该系统。完成这些项目所必要的计算机科学知识在本书中都有介绍，读者唯一需要具备的是一些编程经验。

本书配套的支持网站提供了书中描述的用于构建所有硬件和软件系统所必需的工具和资料，以及用于12个项目的200个测试程序。这些项目和系统可以被修改以满足各种教学需要，我们提供的所有软件都是开源的。

Noam Nisan是Hebrew University of Jerusalem计算机科学与工程研究院的教授。Shimon Schocken是Interdisciplinary Center Herzliya的Efi Arazi计算机科学学院的IDB教授、院长。

封面照片来自Dvora Schocken，1958

图书分类：操作系统

网上订购：www.dearbook.com.cn
第二书店　第一服务

责任编辑：周　筠

ISBN 978-7-121-03336-0

定价：45.00元

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

[General Information]
书名=计算机系统要素 从零开始构建现代计算机
作者=（美）NOAM NISAN，SHIMON SCHOCKEN著
页数=325
SS号=11841471
出版日期=2007.2
出版社=电子工业出版社

封面
书名
版权
前言
目录
介绍：Hello,World Below
第1章　布尔逻辑
1.1　背景知识
1.1.1　布尔代数
1.1.2　门逻辑
1.1.3　实际硬件结构
1.1.4　硬件描述语言（HDL）
1.1.5　硬件仿真
1.2　规范详述
1.2.1　Nand门
1.2.2　基本逻辑门
1.2.3　多位基本门
1.2.4　多通道逻辑门
1.3　实现
1.4　观点
1.5　项目
第2章　布尔运算
2.1　背景知识
2.2　规范详述
2.2.1　加法器
2.2.2　算术逻辑单元（ALU）
2.3　实现
2.4　观点
2.5　项目
第3章　时序逻辑
3.1　背景知识
3.2　规范详述
3.2.1　D触发器
3.2.2　寄存器
3.2.3　存储
3.2.4　计数器

3.3　实现
3.4　观点
3.5　项目
第4章　机器语言
4.1　背景知识
4.1.1　机器
4.1.2　语言
4.1.3　命令
4.2　Hack机器语言规范详述
4.2.1　概述
4.2.2　A-指令
4.2.3　C-指令
4.2.4　符号
4.2.5　输入／输出处理
4.2.6　语法规约和文件格式
4.3　观点
4.4　项目
第5章　计算机体系结构
5.1　背景知识
5.1.1　存储程序概念
5.1.2　冯·诺依曼结构
5.1.3　内存
5.1.4　中央处理器
5.1.5　寄存器
5.1.6　输入和输出
5.2　Hack硬件平台规范详述
5.2.1　概述
5.2.2　中央处理器（CPU）
5.2.3　指令内存
5.2.4　数据内存
5.2.5　计算机
5.3　实现
5.3.1　中央处理器
5.3.2　内存
5.3.3　计算机
5.4　观点

5.5　项目
第6章　汇编编译器
6.1　背景知识
6.2　Hack汇编到二进制的翻译规范详述
6.2.1　语法规约和文件格式
6.2.2　指令
6.2.3　符号
6.2.4　范例
6.3　实现
6.3.1　Parser模块
6.3.2　Code模块
6.3.3　无符号程序的汇编编译器
6.3.4　SymbolTable模块
6.3.5　有符号程序的汇编编译器
6.4　观点
6.5　项目
第7章　虚拟机Ⅰ：堆栈运算
7.1　背景知识
7.1.1　虚拟机范型
7.1.2　堆栈机模型
7.2　VM规范详述，第Ⅰ部分
7.2.1　概论
7.2.2　算术命令和逻辑命令
7.2.3　内存访问命令
7.2.4　程序流程控制命令和函数调用命令
7.2.5　Jack-VM-Hack平台中的程序元素
7.2.6　VM编程实例
7.3　实现
7.3.1　Hack平台上的标准VM映射，第Ⅰ部分
7.3.2　关于VM设计实现的建议
7.3.3　程序结构
7.4　观点
7.5　项目
第8章　虚拟机Ⅱ：程序控制
8.1　背景知识
8.1.1　程序控制流

8.1.2　子程序调用
8.2　VM规范详述，第Ⅱ部分
8.2.1　程序控制流命令
8.2.2　函数调用命令
8.2.3　函数调用协议
8.2.4　初始化
8.3　实现
8.3.1　Hack平台上的标准VM映射，第Ⅱ部分
8.3.2　范例
8.3.3　VM实现的设计建议
8.4　观点
8.5　项目
第9章　高级语言
9.1　背景知识
9.1.1　范例1：Hello World
9.1.2　范例2：过程化编程和数组处理
9.1.3　范例3：抽象数据类型
9.1.4　范例4：链表实现
9.2　Jack语言规范详述
9.2.1　语法要素
9.2.2　程序结构
9.2.3　变量
9.2.4　语句
9.2.5　表达式
9.2.6　子程序调用
9.2.7　Jack标准库
9.3　编写Jack应用程序
9.4　观点
9.5　项目
第10章　编译器Ⅰ：语法分析
10.1　背景知识
10.1.1　词法分析
10.1.2　语法
10.1.3　语法分析（Parsing）
10.2　规范详述
10.2.1　Jack语言语法

10.2.2　Jack语言的语法分析器
10.2.3　语法分析器的输入
10.2.4　语法分析器的输出
10.3　实现
10.3.1　JackAnalyzer模块
10.3.2　JackTokenizer模块
10.3.3　CompilationEngine模块
10.4　观点
10.5　项目
第11章　编译器II：代码生成
11.1　背景知识
11.1.1　数据翻译
11.1.2　命令翻译
11.2　规范详述
11.2.1　虚拟机平台之上的标准映射
11.2.2　编译过程举例
11.3　实现
11.3.1　JackCompiler模块
11.3.2　JackTokenizer模块
11.3.3　SymbolTabel模块
11.3.4　VMWriter模块
11.3.5　CompilationEngine模块
11.4　观点
11.5　项目
第12章　操作系统
12.1　背景知识
12.1.1　数学操作
12.1.2　数字的字符串表示
12.1.3　内存管理
12.1.4　变长数组和字符串
12.1.5　输入／输出管理
12.2　Jack OS规范详述
12.2.1　Math
12.2.2　String
12.2.3　Array
12.2.4　Output

12.2.5 Screen
12.2.6 Keyboard
12.2.7 Memory
12.2.8 Sys
12.3 实现
12.3.1 Math
12.3.2 String
12.3.3 Array
12.3.4 Output
12.3.5 Screen
12.3.6 Keyboard
12.3.7 Memory
12.3.8 Sys
12.4 观点
12.5 项目
第13章 后记：发掘更多乐趣
13.1 硬件的实现
13.2 硬件的改进
13.3 高级语言
13.4 优化
13.5 通信
附录A：硬件描述语言（HDL）
A.1 范例
A.2 约定
A.3 将芯片加载到硬件仿真器
A.4 芯片描述头（接口）
A.5 芯片描述体（实现）
A.5.1 单元
A.5.2 管脚和连接
A.5.3 总线
A.6 内置芯片
A.7 时序芯片
A.7.1 时钟
A.7.2 时钟芯片和管脚
A.7.3 反馈环
A.8 芯片操作的可视化

A.9　已经提供的内置芯片与新的内置芯片
附录B：测试脚本语言
B.1　文件的格式和用法
B.2　在硬件仿真器中测试芯片
B.2.1　范例
B.2.2　数据类型和变量
B.2.3　脚本命令
B.2.4　内置芯片的方法和变量
B.2.5　范例
B.2.6　默认脚本
B.3　在CPU仿真器中测试机器语言程序
B.3.1　范例
B.3.2　变量
B.3.3　命令
B.3.4　默认脚本
B.4　在VM仿真器中测试VM程序
B.4.1　范例
B.4.2　变量
B.4.3　命令
B.4.4　默认脚本
索引